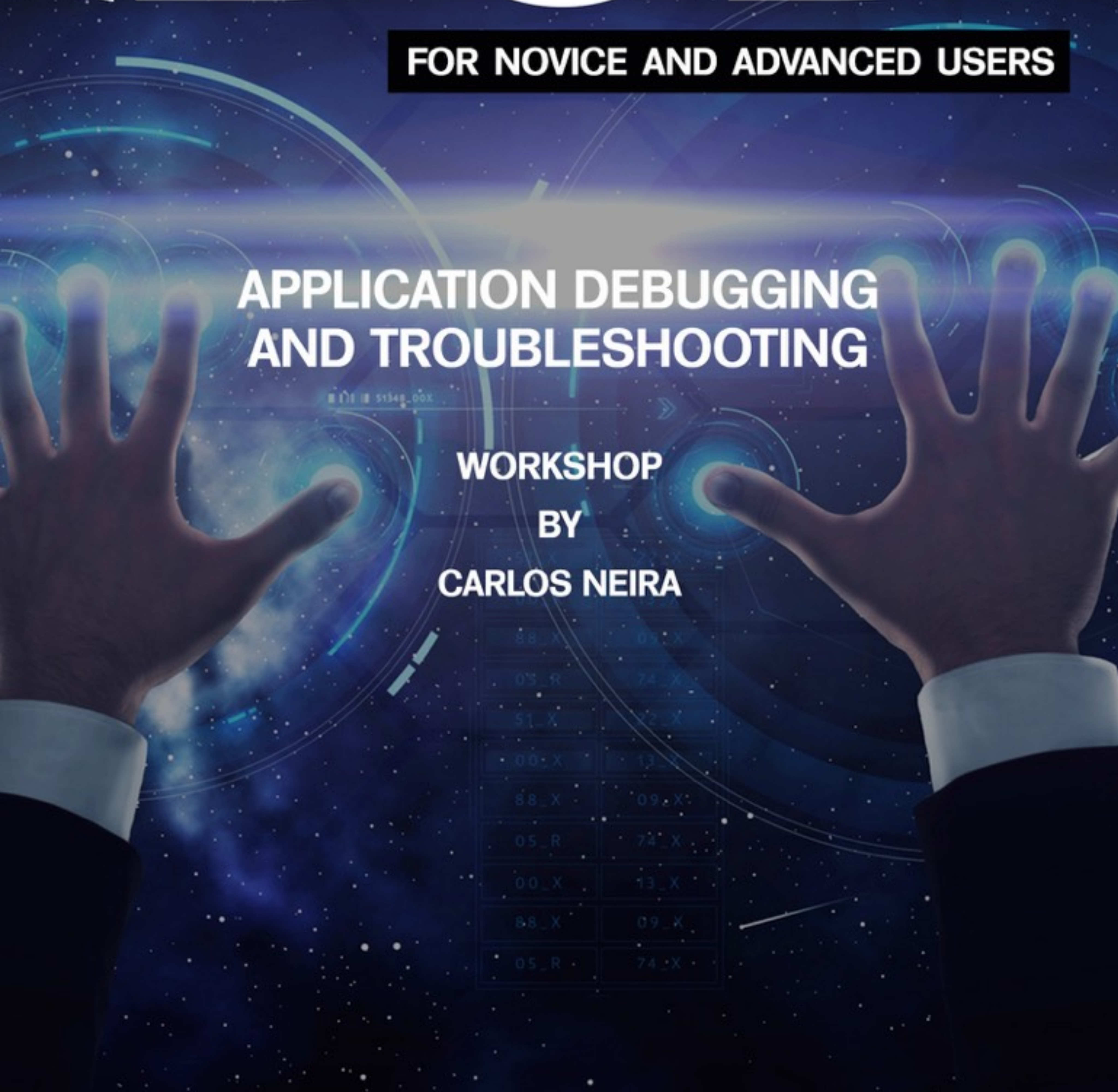
MAGAZINE

BSD

FOR NOVICE AND ADVANCED USERS

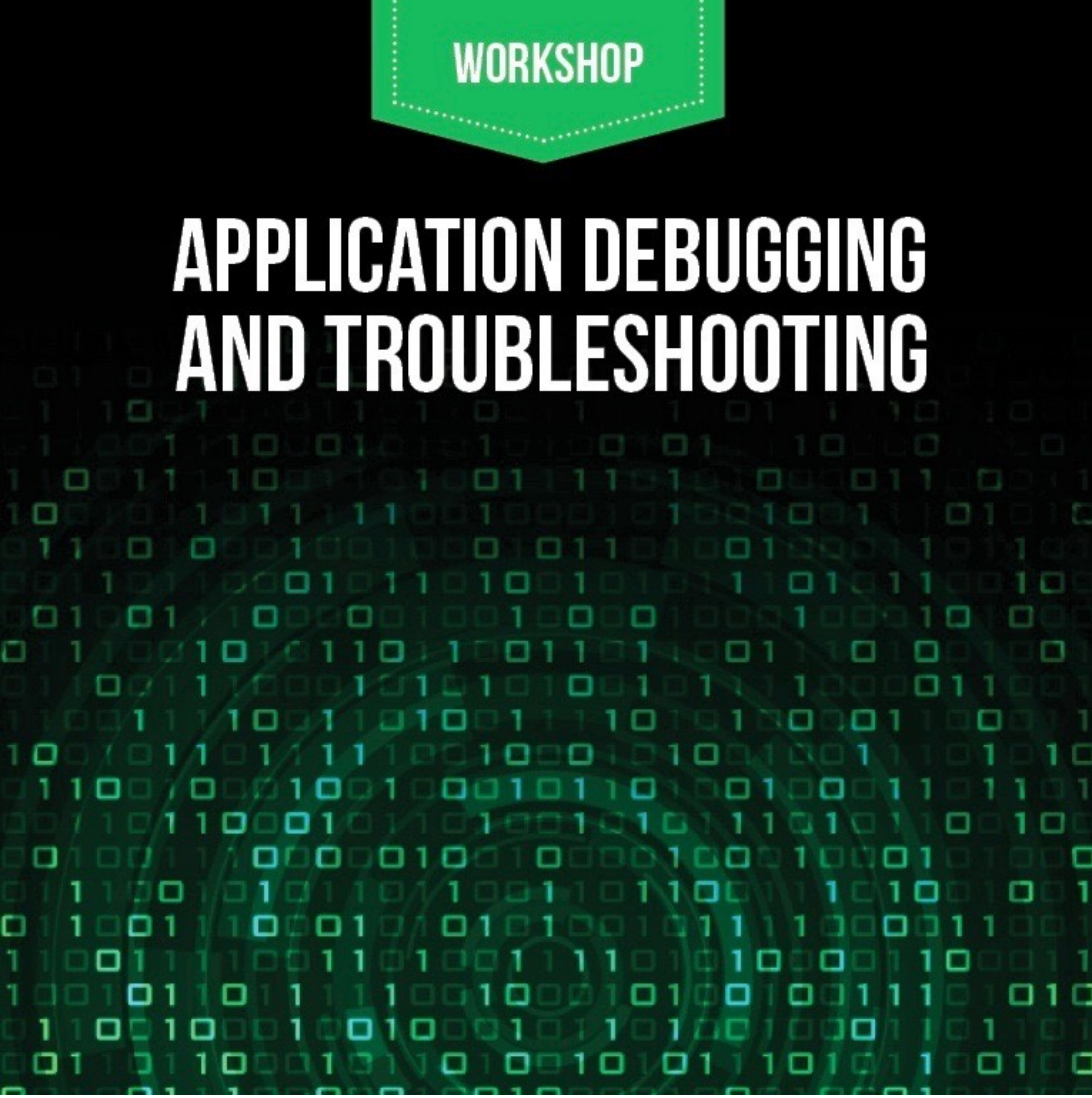


APPLICATION DEBUGGING AND TROUBLESHOOTING

WORKSHOP

BY

CARLOS NEIRA



Start saving your time today and find out how usefull skills you can learn with Carlos.

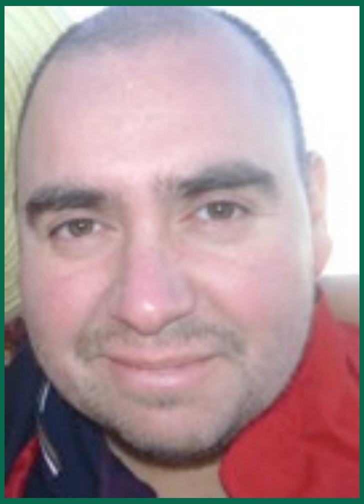
In this workshop you will see real life situations, where debugging skills will save you time, headaches and possibly find a solution with minimal amount of effort.

Debugging/Troubleshooting is a really useful skill when you are working in maintaining legacy applications, doing some small incremental changes to an old code base, where the code has been touched by so many hands over the years and it is becoming really a mess. So, management has decided that the code works as it is and you are not allowed to change it all over “the right way ™”.

INSIDE

- Introduction to the GDB debugger
- Advanced inspection of data structures and variables
- Introduction to the jdb debugger
- Working with core dumps in GDB
- Introduction to Dtrace
- Course Materials

About the Instructor



Carlos Antonio Neira Bustos has worked several years as a C/C++ developer and kernel porting and debugging enterprise legacy applications. He is currently employed as a C developer under Z/OS, debugging and troubleshooting legacy applications for a global financial company. Also he is engaged in independent research on affective computing.

Course format:

- The course is self-paced – you can visit the training whenever you want and your content will be there.
- Once you're in, you keep access forever, even when you finish the course.
- There are no deadlines, except for the ones you set for yourself.
- We designed the course so that a student will need about 18 hours of work to complete the training.
- Your time will be filled with reading, videos, and exercises.

Online: https://bsdmag.org/course/application-debugging-and-troubleshooting-2/

COURSE CURRICULUM

Table of Contents

(05) Introduction to the GDB Debugger

(19) Advanced Inspection of Data Structures and Variables

(27) Introduction to the JDB debugger

(36) Working with Core Dumps in GDB

(47) Introduction to Dtrace

Introduction to the GDB

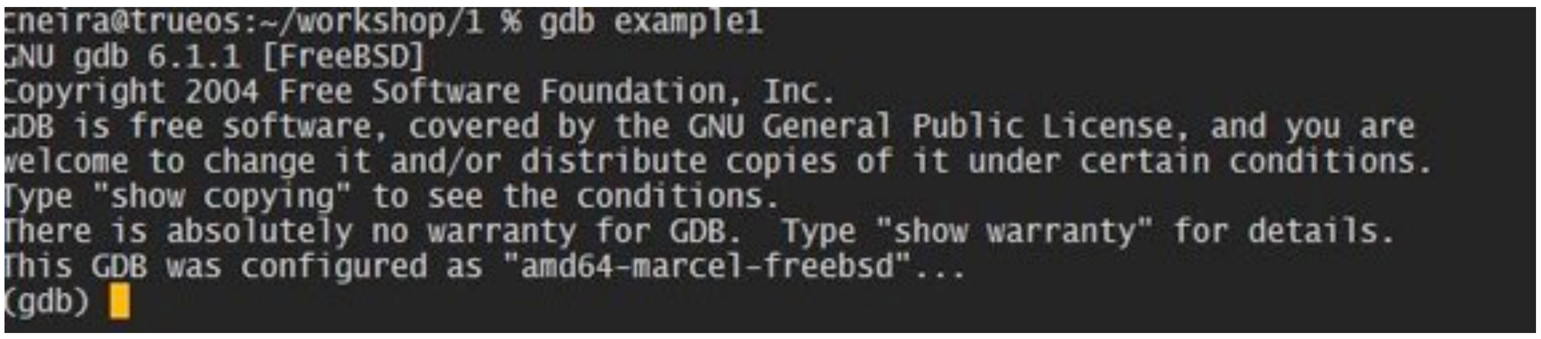
The Basics

To be able to inspect more easily a program, we need to have the symbol table available for the program we intend to debug. This is accomplished using the –g flag of the compiler we are going to use (we could also debug it without the –g flag, but it is really cumbersome sometimes). In our case, we will use FreeBSD 10 as the platform and the clang compiler that comes with it.

After the program has been compiled using the –g flag, we are able to peek inside it using the gdb debugger to start a debugging session. All you need to do is type:

`# gdb <program_name>`

And we will see a (gdb) prompt. That means that we are ready to start typing gdb commands.



```
cneira@trueos:~/workshop/1 % gdb example1
GNU gdb 6.1.1 [FreeBSD]
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "amd64-marcel-freebsd"...
(gdb)
```

Or if the program we need to debug is currently running, we must type:

`#gdb`

`#(gdb) attach <pid of running program>`

Let's start with some basic commands and inspect a running application. For this example, I have selected this application http://freeciv.wikia.com/wiki/Main_Page.

"Freeciv is a Free and Open-Source empire-building strategy game inspired by the history of human civilization. The game commences in prehistory and your mission is to lead your tribe from the Stone Age to the Space Age..."

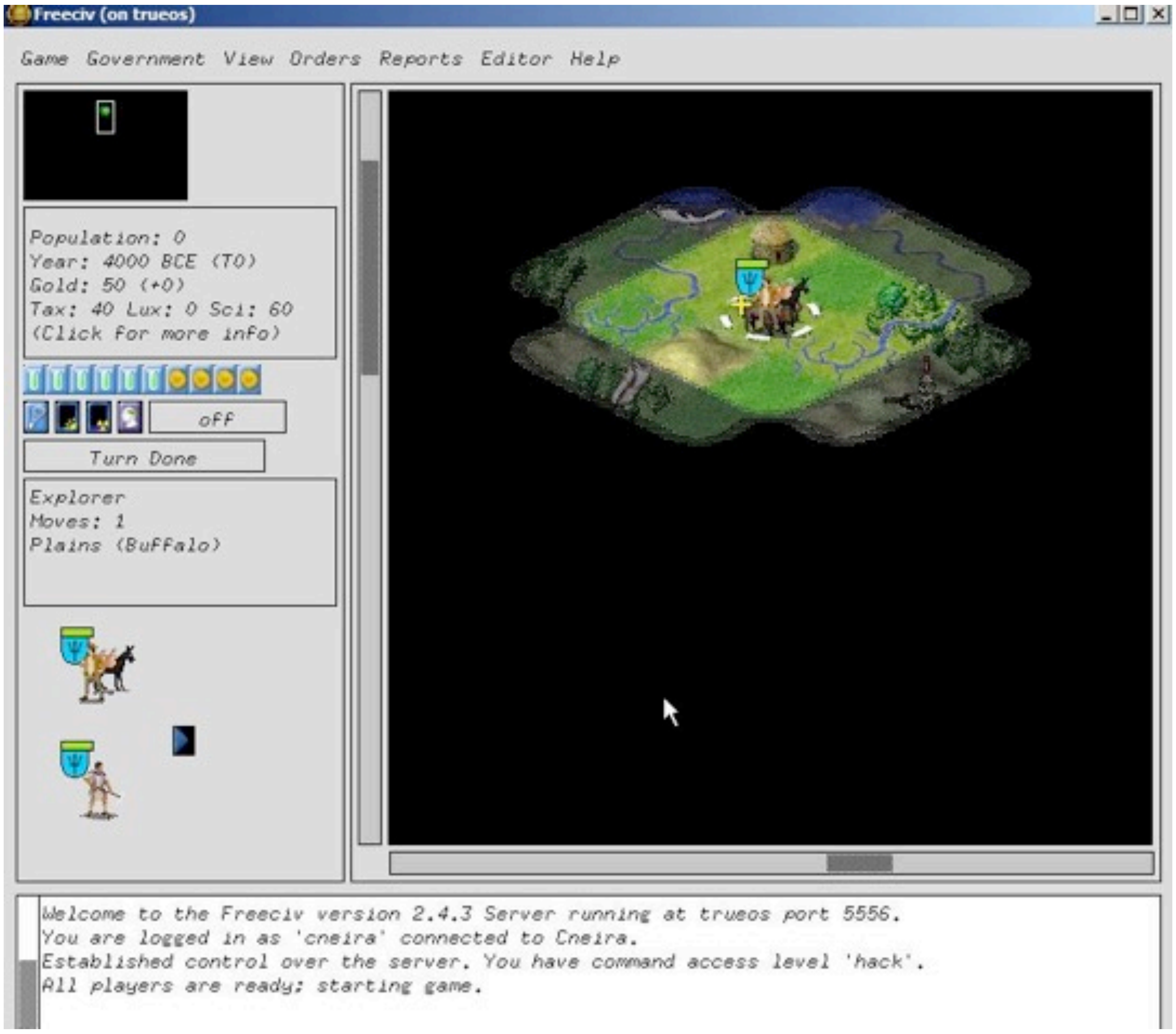
We will inspect the game structures at runtime with gdb. So, let's follow these steps:

Edit /etc/make.conf and add the line WITH_DEBUG=yes (this will not strip your binaries. You will have the symbol table and also add the debug flags to the compiler when compiling the sources of your ports).

Install freeciv from ports.

Start the freeciv server and client (freeciv-server and freeciv-gtk2).

Join your local game.



```
Population: 0
Year: 4000 BCE (T0)
Gold: 50 (+0)
Tax: 40 Lux: 0 Sci: 60
(Click for more info)
```

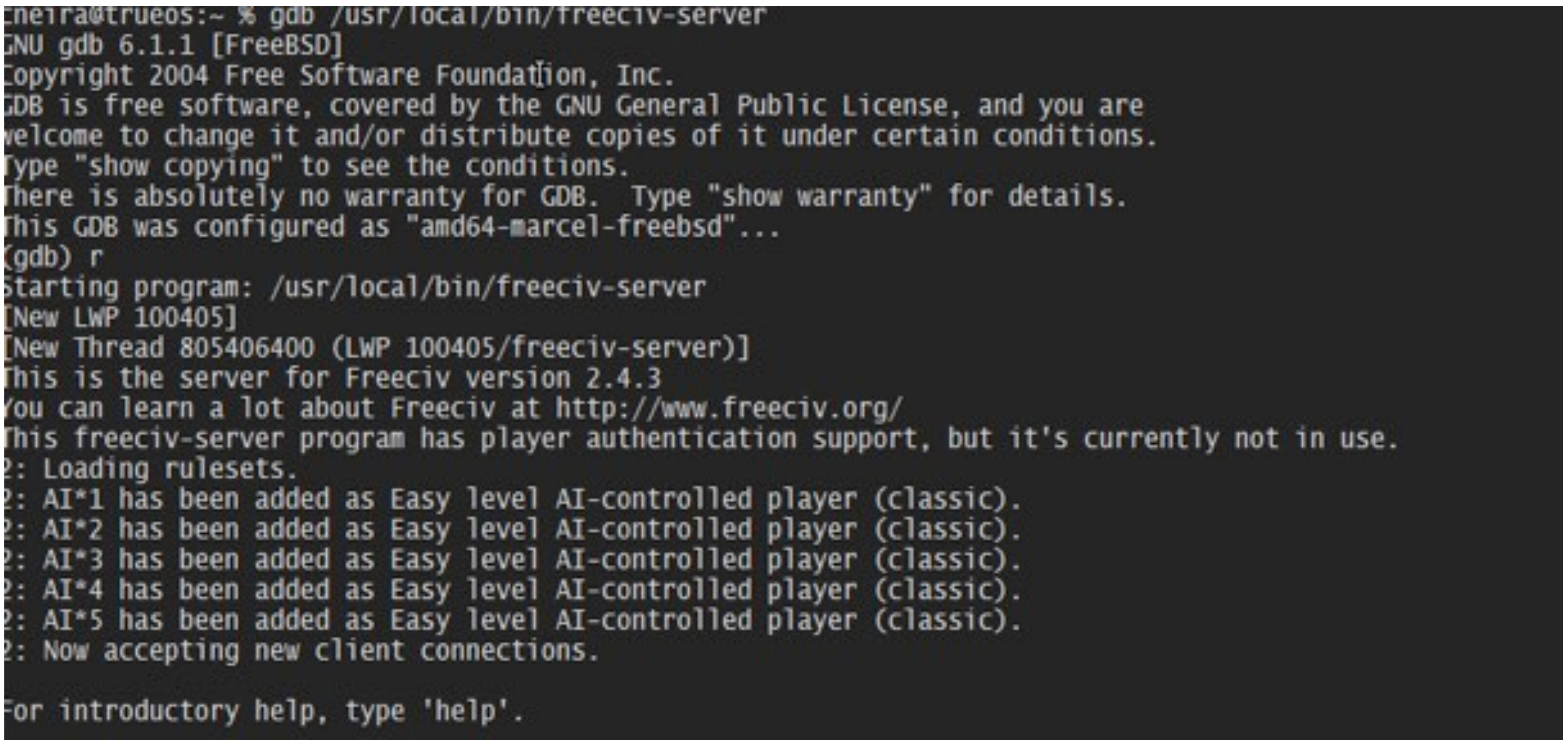
```
Explorer
Moves: 1
Plains (Buffalo)
```

```
Welcome to the Freeciv version 2.4.3 Server running at trueos port 5556.
You are logged in as 'cneira' connected to Cneira.
Established control over the server. You have command access level 'hack'.
All players are ready; starting game.
```

Figure 1. Joined the local game

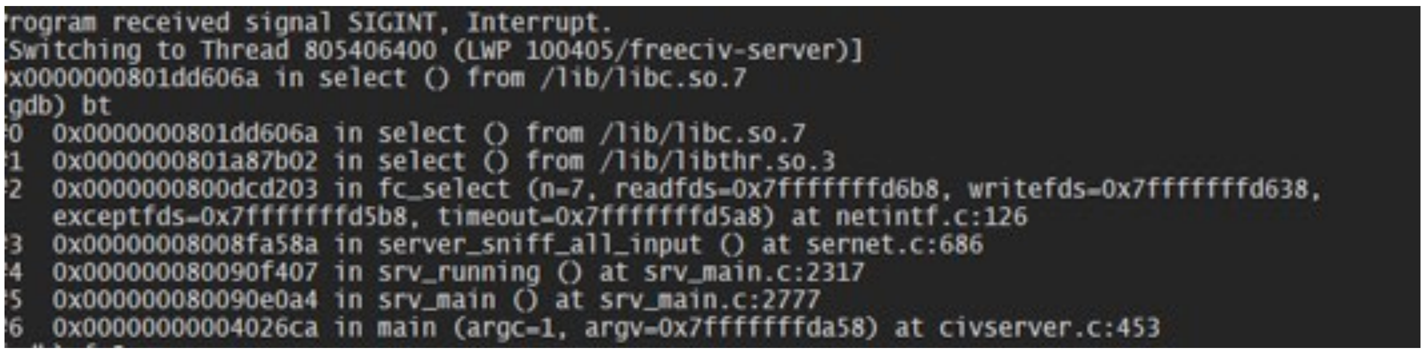
Now, we will use our first gdb command:

```
# gdb /usr/local/bin/freeciv-server
```



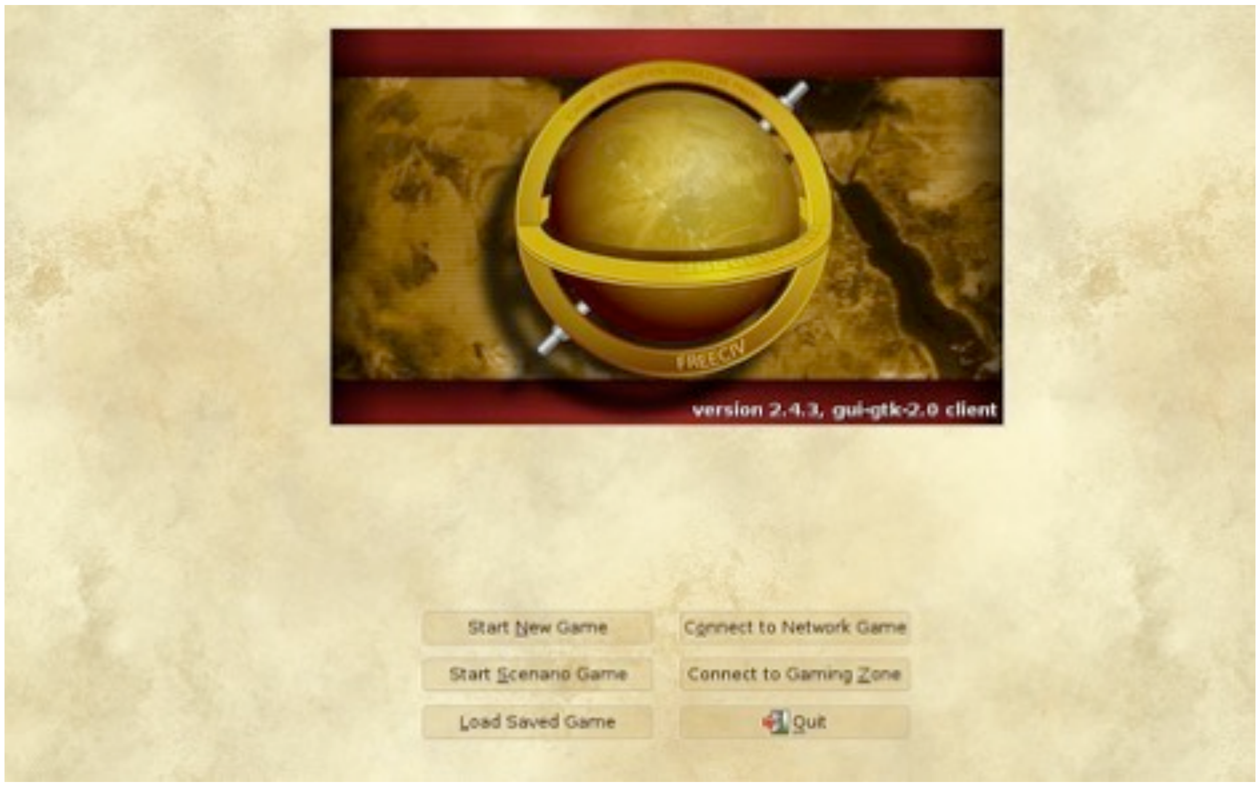
```
neira@trueos:~ % gdb /usr/local/bin/freeciv-server
NU gdb 6.1.1 [FreeBSD]
opyright 2004 Free Software Foundation, Inc.
DB is free software, covered by the GNU General Public License, and you are
elcome to change it and/or distribute copies of it under certain conditions.
ype "show copying" to see the conditions.
here is absolutely no warranty for GDB.  Type "show warranty" for details.
his GDB was configured as "amd64-marcel-freebsd"...
(gdb) r
tarting program: /usr/local/bin/freeciv-server
[New LWP 100405]
[New Thread 805406400 (LWP 100405/freeciv-server)]
his is the server for Freeciv version 2.4.3
ou can learn a lot about Freeciv at http://www.freeciv.org/
his freeciv-server program has player authentication support, but it's currently not in use.
: Loading rulesets.
: AI*1 has been added as Easy level AI-controlled player (classic).
: AI*2 has been added as Easy level AI-controlled player (classic).
: AI*3 has been added as Easy level AI-controlled player (classic).
: AI*4 has been added as Easy level AI-controlled player (classic).
: AI*5 has been added as Easy level AI-controlled player (classic).
: Now accepting new client connections.

or introductory help, type 'help'.
```

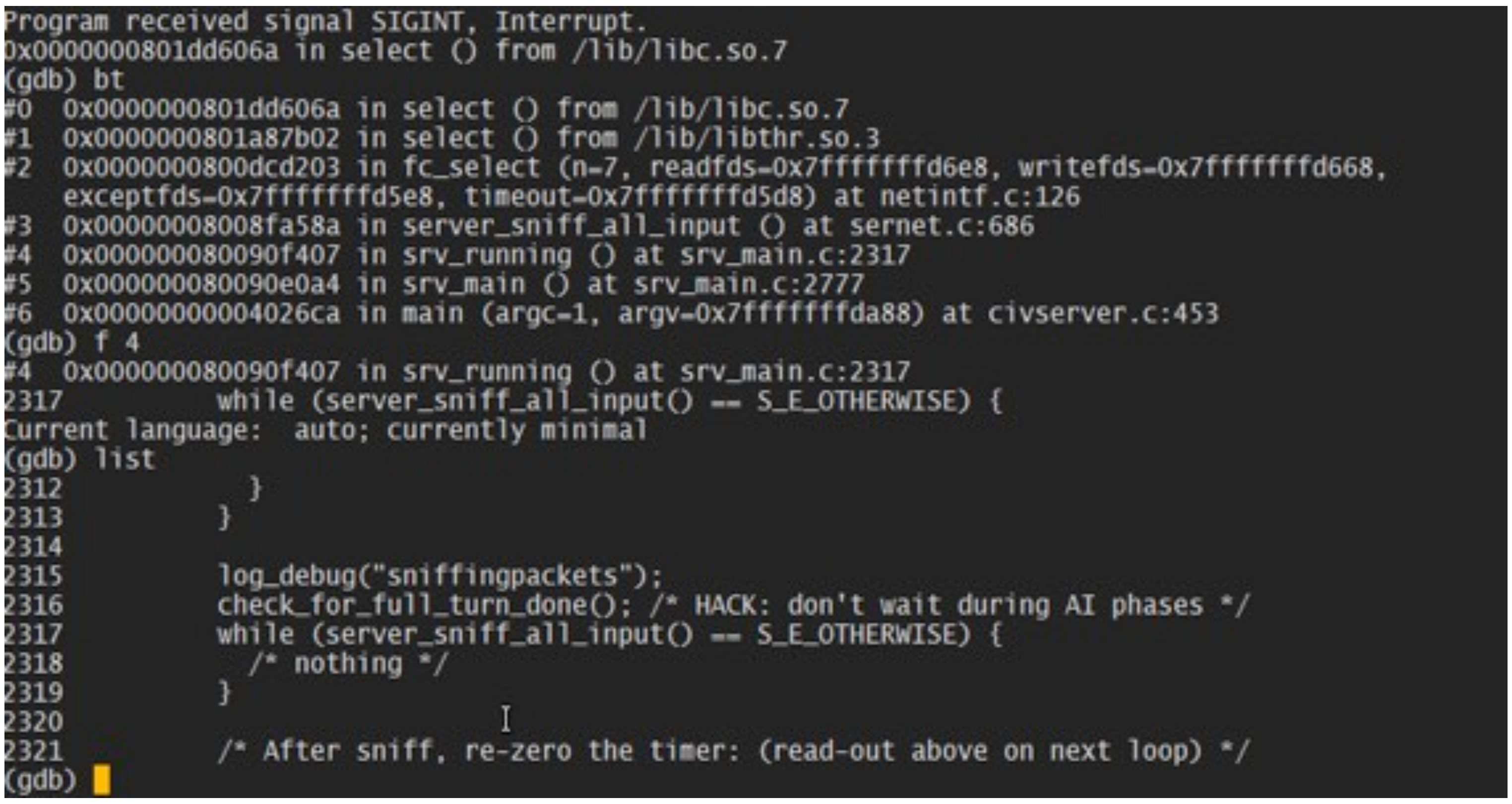
As we don't know anything about how Freeciv works, we will press CTRL-C, this will interrupt the program and we will take from there. For starters, let's interrupt and see where we are. If we want to continue the execution, we type 'continue' or 'c'



```
rogram received signal SIGINT, Interrupt.
[Switching to Thread 805406400 (LWP 100405/freeciv-server)]
x0000000801dd606a in select () from /lib/libc.so.7
(gdb) bt
0  0x0000000801dd606a in select () from /lib/libc.so.7
1  0x0000000801a87b02 in select () from /lib/libthr.so.3
2  0x0000000800dcd203 in fc_select (n=7, readfds=0x7fffffffd6b8, writefds=0x7fffffffd638,
   exceptfds=0x7fffffffd5b8, timeout=0x7fffffffd5a8) at netintf.c:126
3  0x00000008008fa58a in server_sniff_all_input () at sernet.c:686
4  0x000000080090f407 in srv_running () at srv_main.c:2317
5  0x000000080090e0a4 in srv_main () at srv_main.c:2777
6  0x00000000004026ca in main (argc=1, argv=0x7fffffffda58) at civserver.c:453
```

Here is a screenshot from the client program freeciv-gtk2. We need to join our local game as we are going to debug the server.

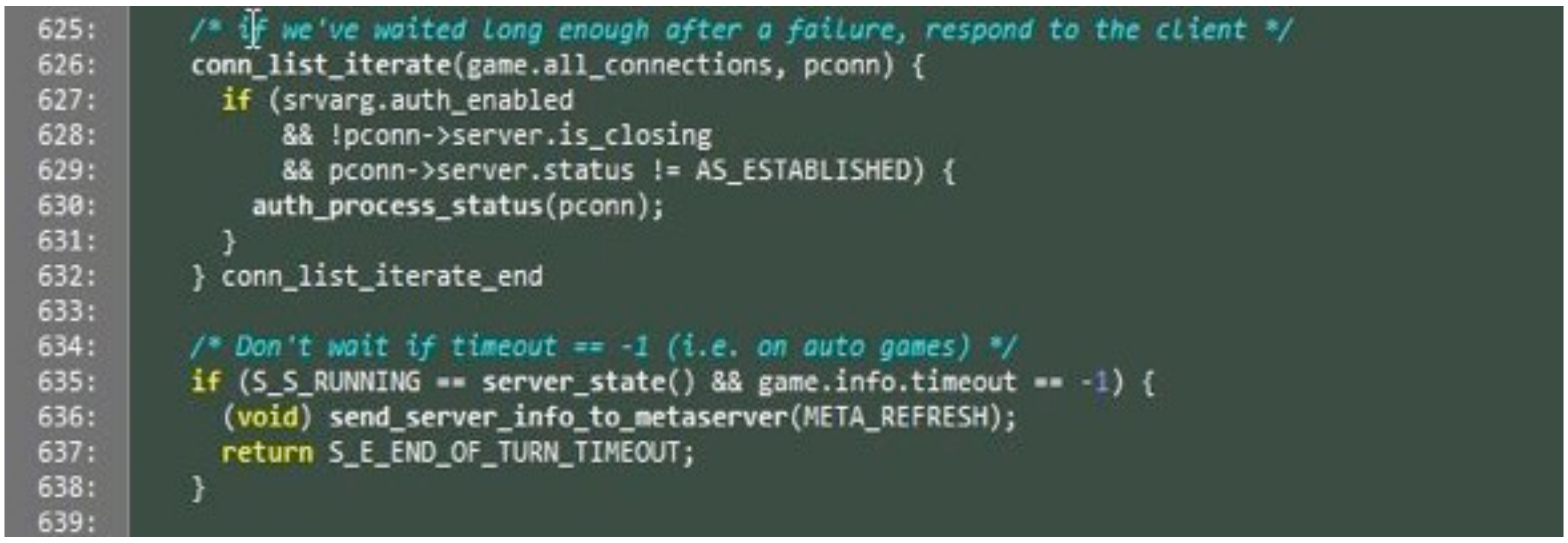




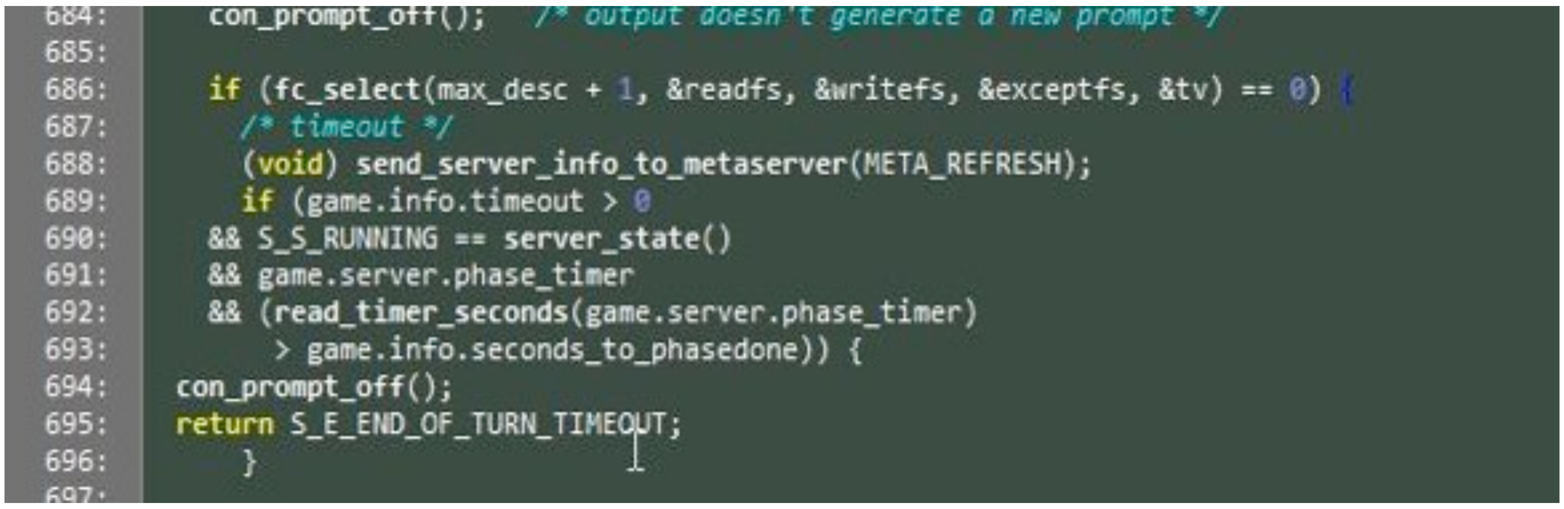
```
rogram received signal SIGINT, Interrupt.
x0000000801dd606a in select () from /lib/libc.so.7
(gdb) bt
#0  0x0000000801dd606a in select () from /lib/libc.so.7
#1  0x0000000801a87b02 in select () from /lib/libthr.so.3
#2  0x0000000800dcd203 in fc_select (n=7, readfds=0x7fffffffd6e8, writefds=0x7fffffffd668,
    exceptfds=0x7fffffffd5e8, timeout=0x7fffffffd5d8) at netintf.c:126
#3  0x00000008008fa58a in server_sniff_all_input () at sernet.c:686
#4  0x000000080090f407 in srv_running () at srv_main.c:2317
#5  0x000000080090e0a4 in srv_main () at srv_main.c:2777
#6  0x00000000004026ca in main (argc=1, argv=0x7fffffffda88) at civserver.c:453
(gdb) f 4
#4  0x000000080090f407 in srv_running () at srv_main.c:2317
2317          while (server_sniff_all_input() == S_E_OTHERWISE) {
urrent language:  auto; currently minimal
(gdb) list
2312            }
2313          }
2314
2315          log_debug("sniffingpackets");
2316          check_for_full_turn_done(); /* HACK: don't wait during AI phases */
2317          while (server_sniff_all_input() == S_E_OTHERWISE) {
2318            /* nothing */
2319          }
2320
2321          /* After sniff, re-zero the timer: (read-out above on next loop) */
(gdb) 
```

The #<num> you see are the stackframes, or simply called frames. When your program is started, the stack has only one frame, that of the function main. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is created. Each time a function returns, frame for that function invocation is eliminated. If a function is recursive, there could be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

Let's go into frame 3. To do this, we type either 'frame 3' or 'f 3'.



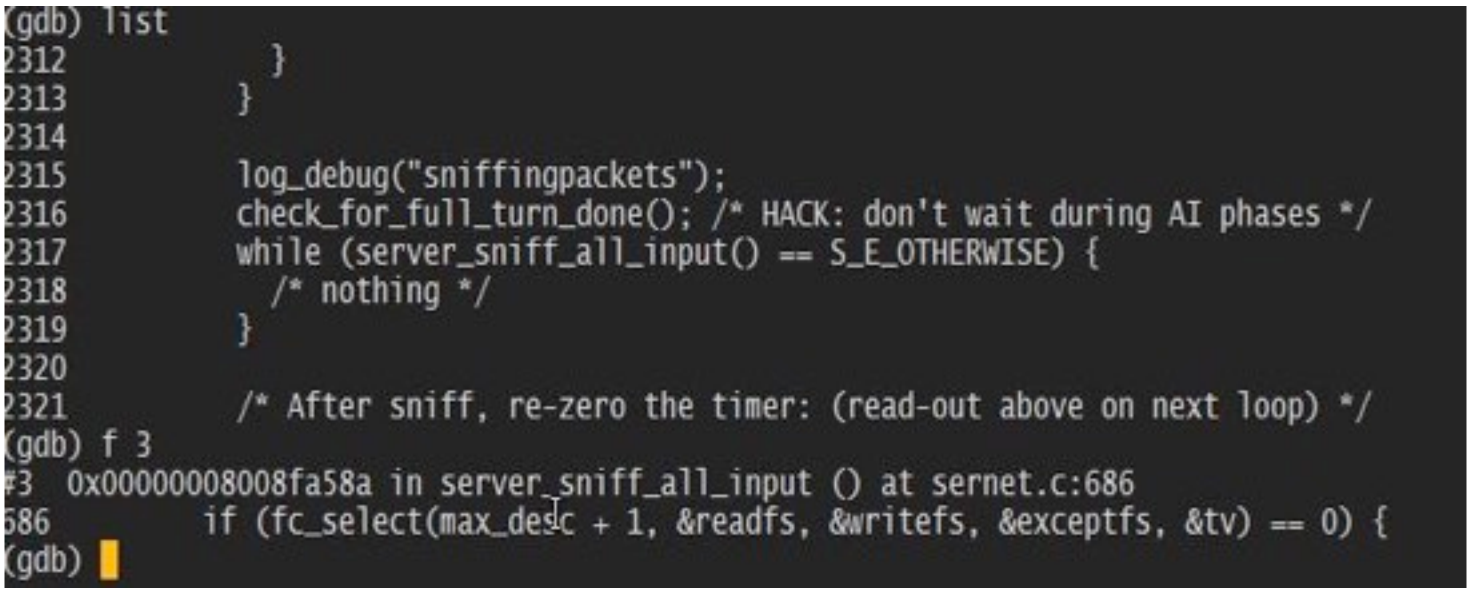
```
  /* if we've waited long enough after a failure, respond to the client */
  conn_list_iterate(game.all_connections, pconn) {
    if (srvarg.auth_enabled
        && !pconn->server.is_closing
        && pconn->server.status != AS_ESTABLISHED) {
      auth_process_status(pconn);
    }
  } conn_list_iterate_end

  /* Don't wait if timeout == -1 (i.e. on auto games) */
  if (S_S_RUNNING == server_state() && game.info.timeout == -1) {
    (void) send_server_info_to_metaserver(META_REFRESH);
    return S_E_END_OF_TURN_TIMEOUT;
  }

```



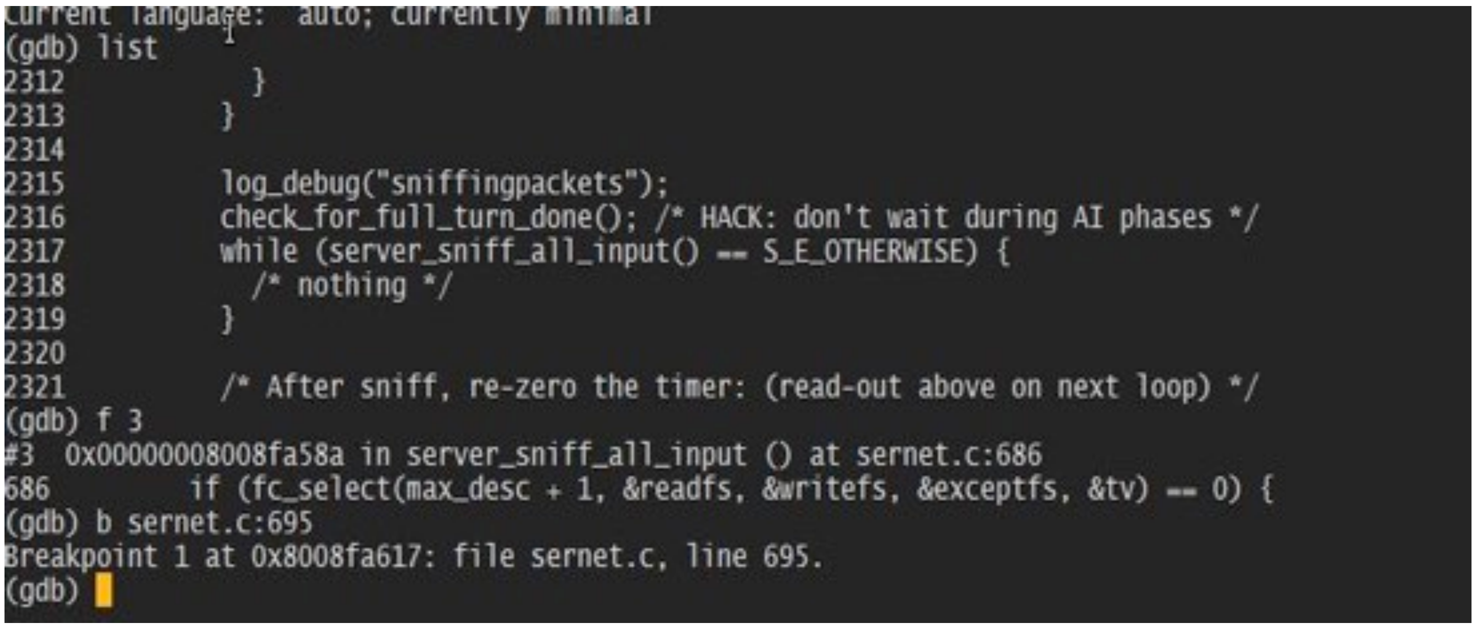
```
    con_prompt_off();   /* output doesn't generate a new prompt */

    if (fc_select(max_desc + 1, &readfs, &writefs, &exceptfs, &tv) == 0) {
      /* timeout */
      (void) send_server_info_to_metaserver(META_REFRESH);
      if (game.info.timeout > 0
    && S_S_RUNNING == server_state()
    && game.server.phase_timer
    && (read_timer_seconds(game.server.phase_timer)
        > game.info.seconds_to_phasedone)) {
  con_prompt_off();
  return S_E_END_OF_TURN_TIMEOUT;
      }
```

It seems that the server will send us end of turn. Let's make sure a break point is set as follows:

```
 the format is >

<break|b> <source.c>:<line number> (gdb) b sernet.c:695
```

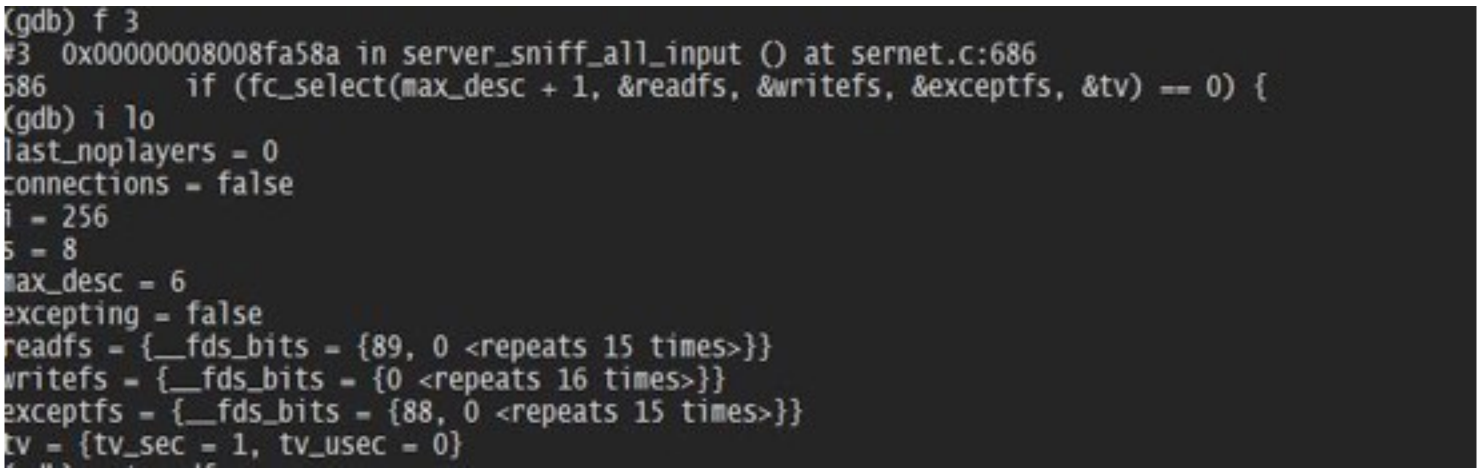


```
(gdb) list
2312              }
2313            }
2314
2315            log_debug("sniffingpackets");
2316            check_for_full_turn_done(); /* HACK: don't wait during AI phases */
2317            while (server_sniff_all_input() == S_E_OTHERWISE) {
2318              /* nothing */
2319            }
2320
2321            /* After sniff, re-zero the timer: (read-out above on next loop) */
(gdb) f 3
#3  0x00000008008fa58a in server_sniff_all_input () at sernet.c:686
686         if (fc_select(max_desc + 1, &readfs, &writefs, &exceptfs, &tv) == 0) {
(gdb) 
```



```
Current language:  auto; currently minimal
(gdb) list
2312              }
2313            }
2314
2315            log_debug("sniffingpackets");
2316            check_for_full_turn_done(); /* HACK: don't wait during AI phases */
2317            while (server_sniff_all_input() == S_E_OTHERWISE) {
2318              /* nothing */
2319            }
2320
2321            /* After sniff, re-zero the timer: (read-out above on next loop) */
(gdb) f 3
#3  0x00000008008fa58a in server_sniff_all_input () at sernet.c:686
686         if (fc_select(max_desc + 1, &readfs, &writefs, &exceptfs, &tv) == 0) {
(gdb) b sernet.c:695
Breakpoint 1 at 0x8008fa617: file sernet.c, line 695.
(gdb) 
```

It seems we are wrong. Let's interrupt again and inspect the data at this point.

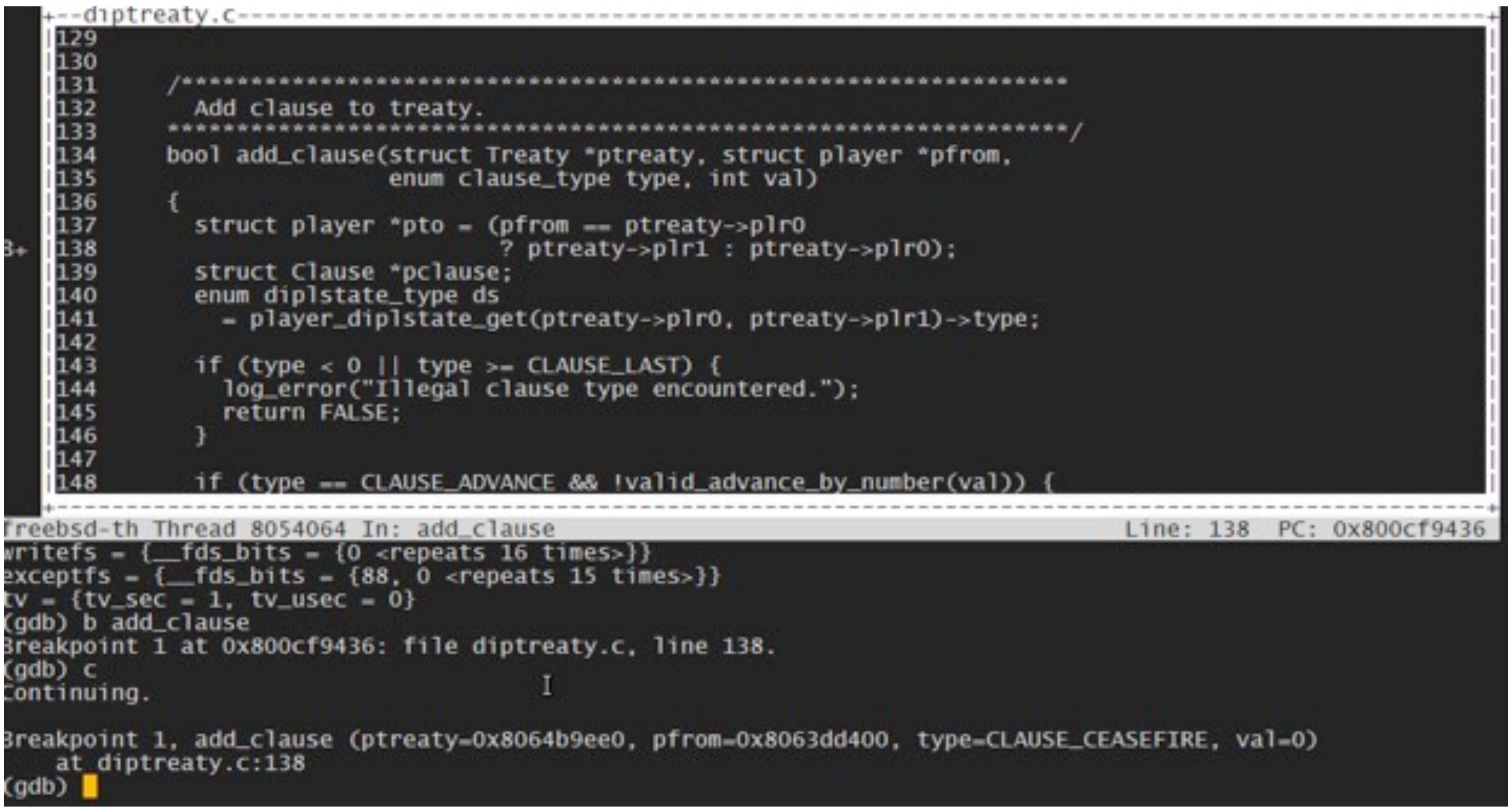


```
(gdb) f 3
#3  0x00000008008fa58a in server_sniff_all_input () at sernet.c:686
686         if (fc_select(max_desc + 1, &readfs, &writefs, &exceptfs, &tv) == 0) {
(gdb) i lo
last_noplayers = 0
connections = false
i = 256
s = 8
max_desc = 6
excepting = false
readfs = {__fds_bits = {89, 0 <repeats 15 times>}}
writefs = {__fds_bits = {0 <repeats 16 times>}}
exceptfs = {__fds_bits = {88, 0 <repeats 15 times>}}
tv = {tv_sec = 1, tv_usec = 0}
```

Typing **i lo** means info locals which will display all local variables in this frame and their values, which is pretty handy.

Let’s take a look at something easier to see. Sometimes in freeciv, another civilization will try to enter into negotiating terms with us. Therefore, looking at the source code, we find the add_clause function in the diptreaty.c source code. That function will add a term which will make the other part accept or reject our terms.

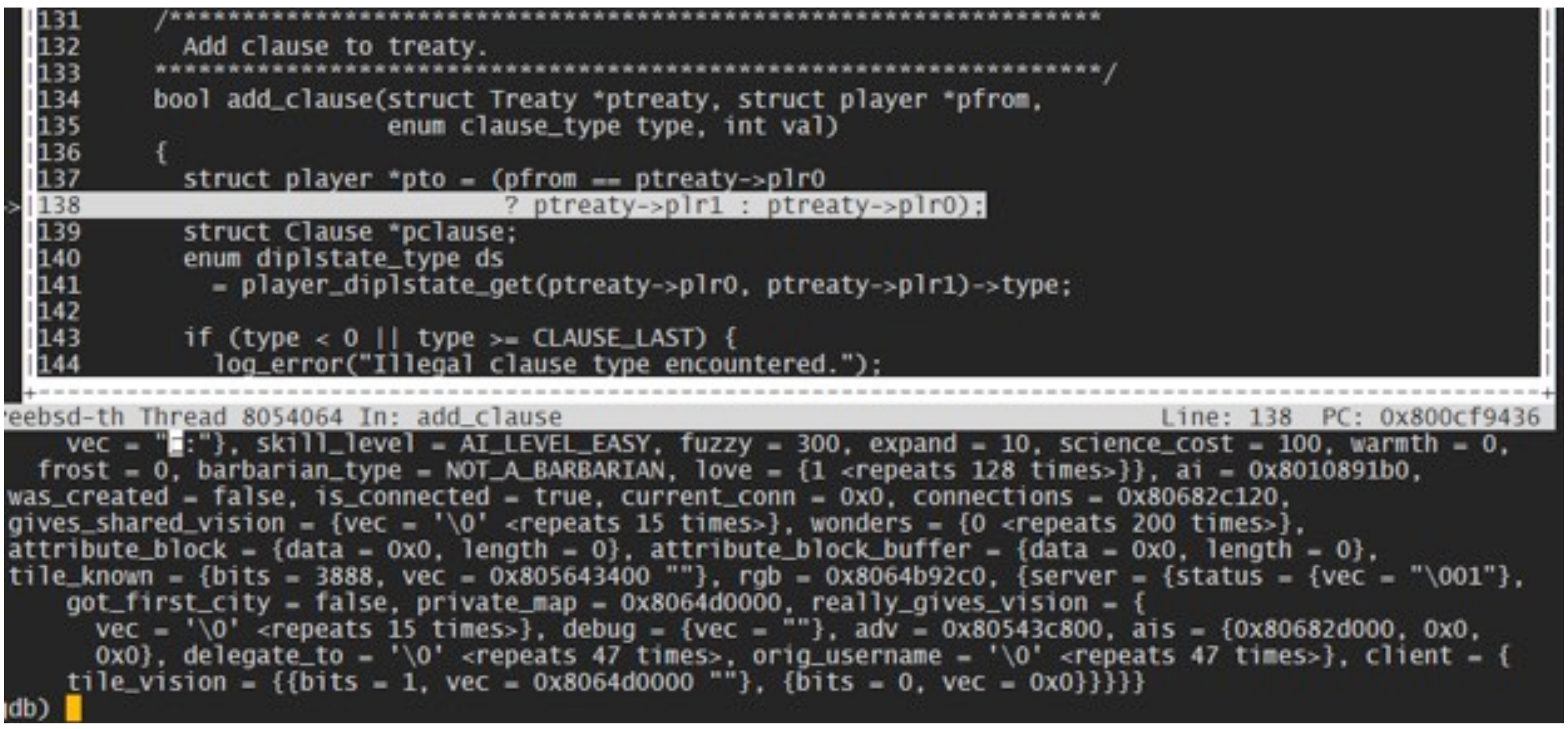
```
/usr/home/cneira/workshop/freeciv-2.4.3/common
```



```
+--diptreaty.c---------------------------------------------------------------+


    /*****************************************************************
      Add clause to treaty.
    *****************************************************************/
    bool add_clause(struct Treaty *ptreaty, struct player *pfrom,
                    enum clause_type type, int val)
    {
      struct player *pto = (pfrom == ptreaty->plr0
                            ? ptreaty->plr1 : ptreaty->plr0);
      struct Clause *pclause;
      enum diplstate_type ds
        = player_diplstate_get(ptreaty->plr0, ptreaty->plr1)->type;

      if (type < 0 || type >= CLAUSE_LAST) {
        log_error("Illegal clause type encountered.");
        return FALSE;
      }

      if (type == CLAUSE_ADVANCE && !valid_advance_by_number(val)) {
+----------------------------------------------------------------------------+
freebsd-th Thread 8054064 In: add_clause                 Line: 138   PC: 0x800cf9436
writefs = {__fds_bits = {0 <repeats 16 times>}}
exceptfs = {__fds_bits = {88, 0 <repeats 15 times>}}
tv = {tv_sec = 1, tv_usec = 0}
(gdb) b add_clause
Breakpoint 1 at 0x800cf9436: file diptreaty.c, line 138.
(gdb) c
Continuing.

Breakpoint 1, add_clause (ptreaty=0x8064b9ee0, pfrom=0x8063dd400, type=CLAUSE_CEASEFIRE, val=0)
    at diptreaty.c:138
(gdb)
```

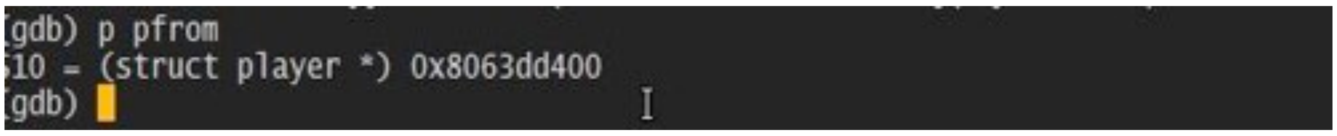
After playing a few minutes, we hit a break point. At this point, we don’t even know which civilization has approached us to negotiate terms.

Now, we could know ahead since we set the breakpoint where the negotiation starts.



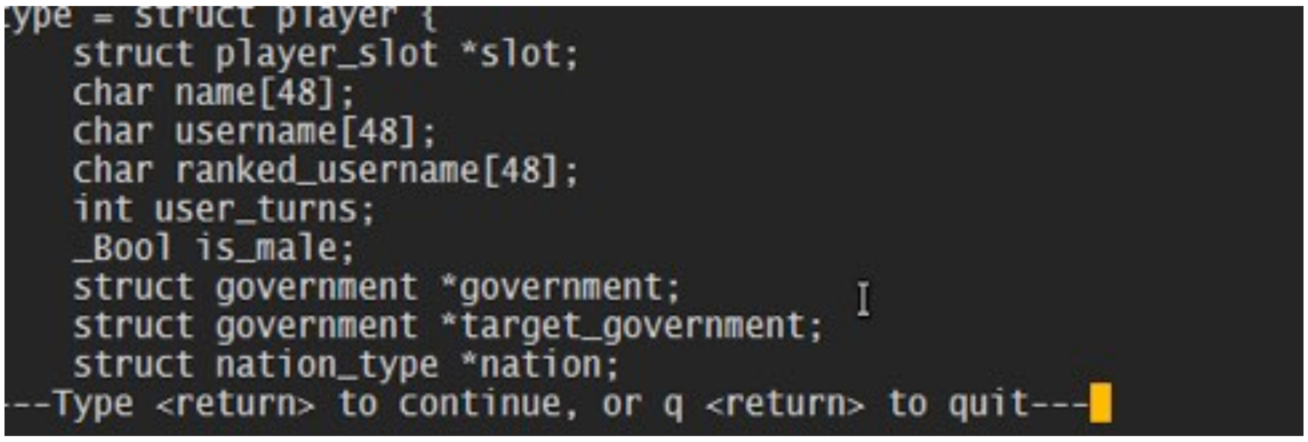
```
    /*****************************************************************
      Add clause to treaty.
    *****************************************************************/
    bool add_clause(struct Treaty *ptreaty, struct player *pfrom,
                    enum clause_type type, int val)
    {
      struct player *pto = (pfrom == ptreaty->plr0
                            ? ptreaty->plr1 : ptreaty->plr0);
      struct Clause *pclause;
      enum diplstate_type ds
        = player_diplstate_get(ptreaty->plr0, ptreaty->plr1)->type;

      if (type < 0 || type >= CLAUSE_LAST) {
        log_error("Illegal clause type encountered.");
+----------------------------------------------------------------------------+
eebsd-th Thread 8054064 In: add_clause                   Line: 138   PC: 0x800cf9436
    vec = " :"}, skill_level = AI_LEVEL_EASY, fuzzy = 300, expand = 10, science_cost = 100, warmth = 0,
  frost = 0, barbarian_type = NOT_A_BARBARIAN, love = {1 <repeats 128 times>}}, ai = 0x8010891b0,
was_created = false, is_connected = true, current_conn = 0x0, connections = 0x80682c120,
gives_shared_vision = {vec = '\0' <repeats 15 times>}, wonders = {0 <repeats 200 times>},
attribute_block = {data = 0x0, length = 0}, attribute_block_buffer = {data = 0x0, length = 0},
tile_known = {bits = 3888, vec = 0x805643400 ""}, rgb = 0x8064b92c0, {server = {status = {vec = "\001"},
    got_first_city = false, private_map = 0x8064d0000, really_gives_vision = {
      vec = '\0' <repeats 15 times>}, debug = {vec = ""}, adv = 0x80543c800, ais = {0x80682d000, 0x0,
      0x0}, delegate_to = '\0' <repeats 47 times>, orig_username = '\0' <repeats 47 times>}, client = {
    tile_vision = {{bits = 1, vec = 0x8064d0000 ""}, {bits = 0, vec = 0x0}}}}}
db)
```

I assume the negotiation civilization should be in the pfrom pointer.



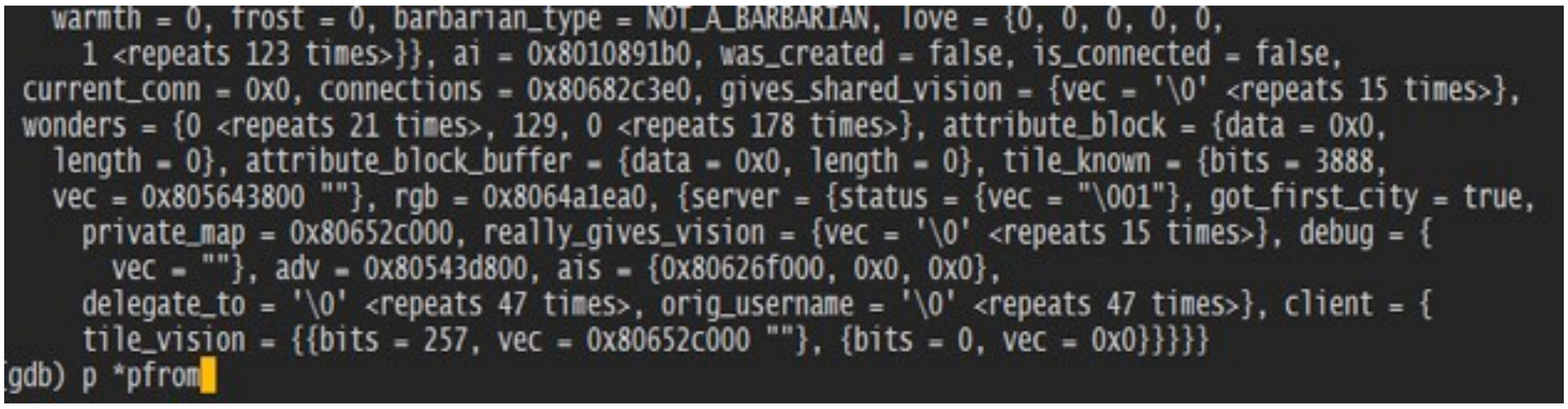
```
(gdb) p pfrom
$10 = (struct player *) 0x8063dd400
(gdb)
```

To print the variables values, we just type ‘p’. In this case, ‘p’ is a pointer to a player structure. If we would like to check the definition of the player structure, we just type ptype pfrom, and the structure definition will be displayed.



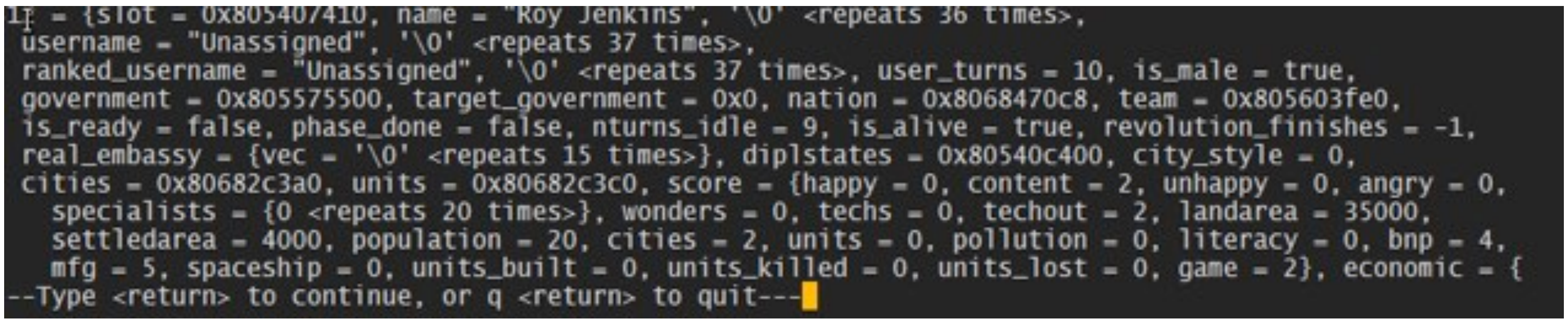
```
type = struct player {
    struct player_slot *slot;
    char name[48];
    char username[48];
    char ranked_username[48];
    int user_turns;
    _Bool is_male;
    struct government *government;
    struct government *target_government;
    struct nation_type *nation;
--Type <return> to continue, or q <return> to quit---
```

Now, let’s find out the values of these fields that the civilization demands. Since the ***pfrom*** is a pointer, we need to use pointer notation to check its contents.



```
   warmth = 0, frost = 0, barbarian_type = NOT_A_BARBARIAN, love = {0, 0, 0, 0, 0,
     1 <repeats 123 times>}}, ai = 0x8010891b0, was_created = false, is_connected = false,
  current_conn = 0x0, connections = 0x80682c3e0, gives_shared_vision = {vec = '\0' <repeats 15 times>},
  wonders = {0 <repeats 21 times>, 129, 0 <repeats 178 times>}, attribute_block = {data = 0x0,
    length = 0}, attribute_block_buffer = {data = 0x0, length = 0}, tile_known = {bits = 3888,
    vec = 0x805643800 ""}, rgb = 0x8064a1ea0, {server = {status = {vec = "\001"}, got_first_city = true,
      private_map = 0x80652c000, really_gives_vision = {vec = '\0' <repeats 15 times>}, debug = {
        vec = ""}, adv = 0x80543d800, ais = {0x80626f000, 0x0, 0x0},
      delegate_to = '\0' <repeats 47 times>, orig_username = '\0' <repeats 47 times>}, client = {
      tile_vision = {{bits = 257, vec = 0x80652c000 ""}, {bits = 0, vec = 0x0}}}}}
(gdb) p *pfrom
```

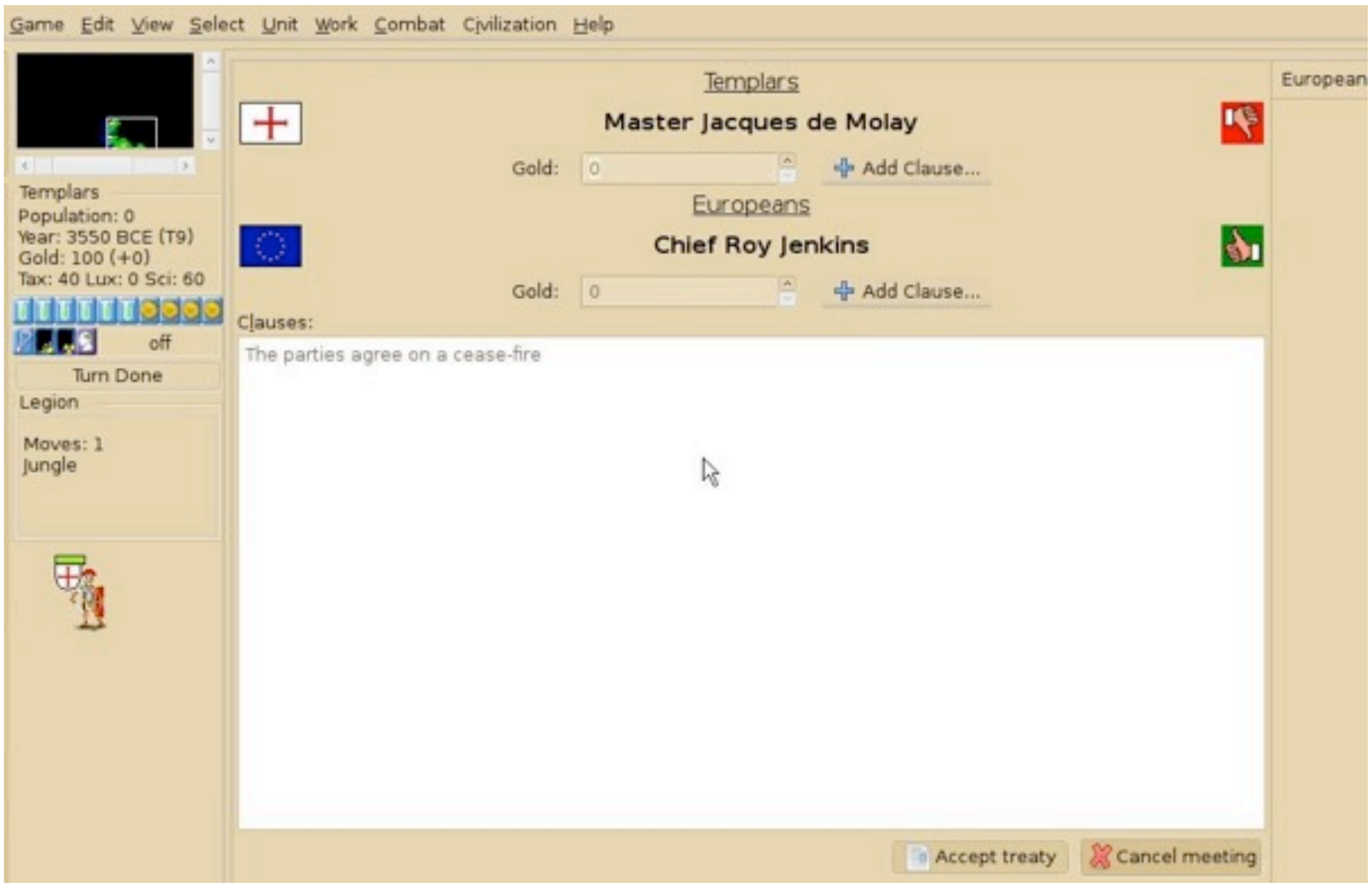
And there we go! The full dump for the player ***struct***



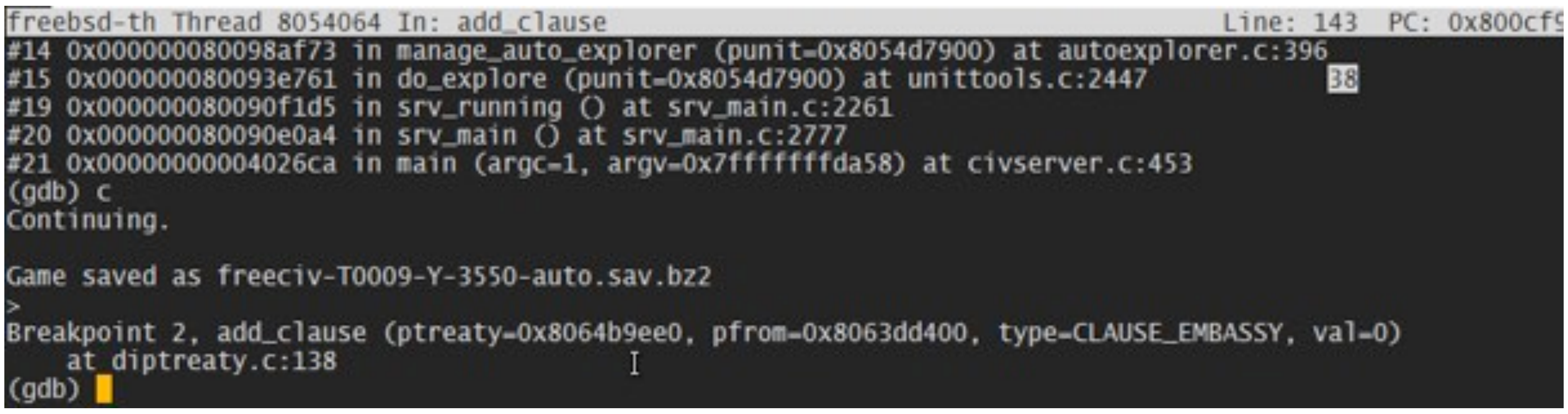
```
$11 = {slot = 0x805407410, name = "Roy Jenkins", '\0' <repeats 36 times>,
  username = "Unassigned", '\0' <repeats 37 times>,
  ranked_username = "Unassigned", '\0' <repeats 37 times>, user_turns = 10, is_male = true,
  government = 0x805575500, target_government = 0x0, nation = 0x8068470c8, team = 0x805603fe0,
  is_ready = false, phase_done = false, nturns_idle = 9, is_alive = true, revolution_finishes = -1,
  real_embassy = {vec = '\0' <repeats 15 times>}, diplstates = 0x80540c400, city_style = 0,
  cities = 0x80682c3a0, units = 0x80682c3c0, score = {happy = 0, content = 2, unhappy = 0, angry = 0,
    specialists = {0 <repeats 20 times>}, wonders = 0, techs = 0, techout = 2, landarea = 35000,
    settledarea = 4000, population = 20, cities = 2, units = 0, pollution = 0, literacy = 0, bnp = 4,
    mfg = 5, spaceship = 0, units_built = 0, units_killed = 0, units_lost = 0, game = 2}, economic = {
--Type <return> to continue, or q <return> to quit---
```

Looking at the player ***struct ,*** it seems that the leader’s name is Roy Jenkins, and looking at the ***backtrace (bt),*** the clause of the treaty seems to be “cease fire”. So, we are going to be offered a peace treaty.

To continue executing the program, type ‘next’ or ‘n’ . Something like this will be displayed in the diplomacy tab:

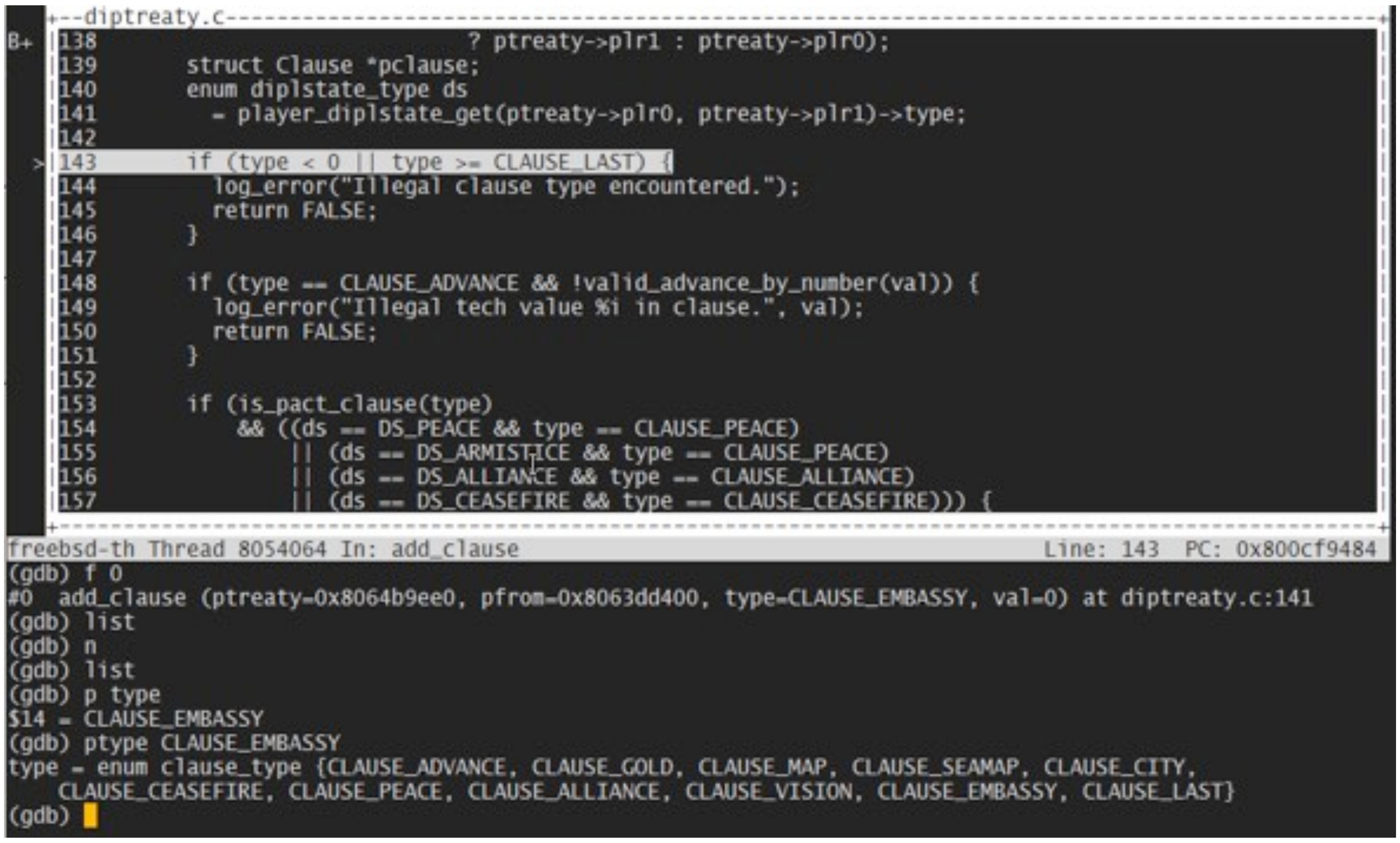


What you cannot see in the screenshot is that I have requested an embassy in return for the cease-fire treaty, but here it is:

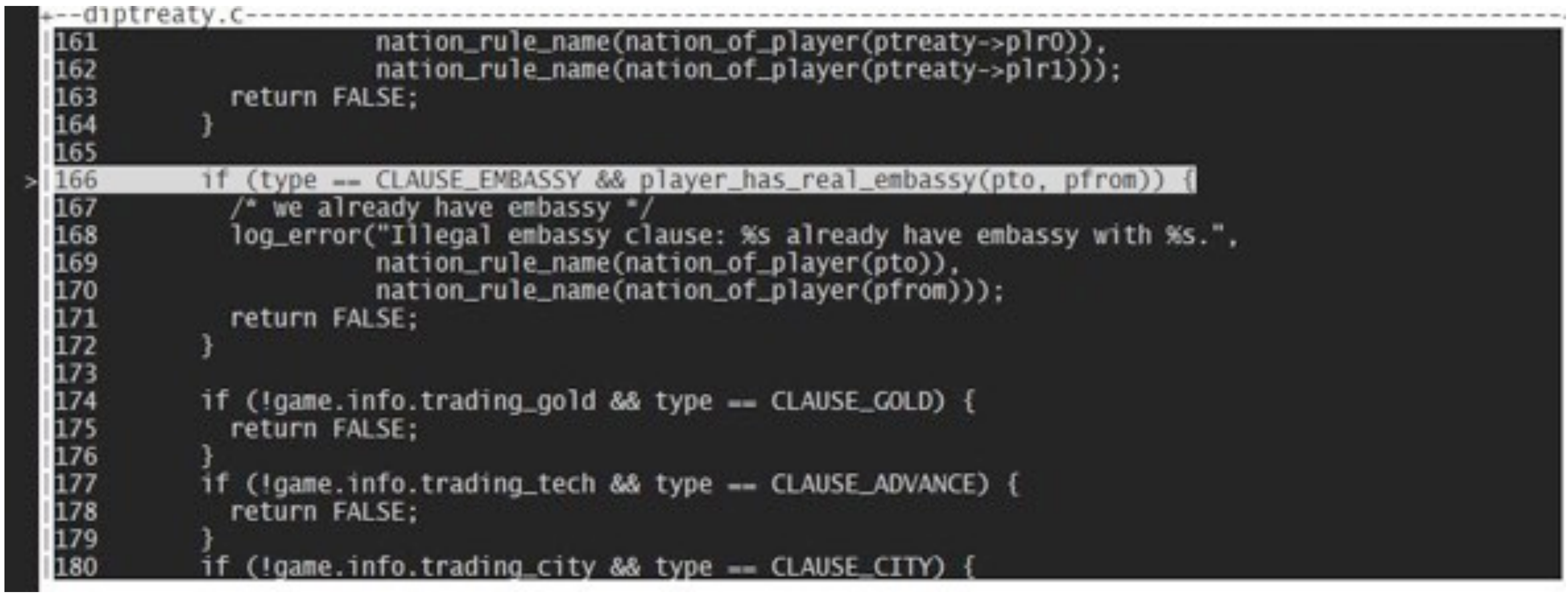


```
freebsd-th Thread 8054064 In: add_clause                                   Line: 143   PC: 0x800cf9
#14 0x000000080098af73 in manage_auto_explorer (punit=0x8054d7900) at autoexplorer.c:396
#15 0x000000080093e761 in do_explore (punit=0x8054d7900) at unittools.c:2447          38
#19 0x000000080090f1d5 in srv_running () at srv_main.c:2261
#20 0x000000080090e0a4 in srv_main () at srv_main.c:2777
#21 0x00000000004026ca in main (argc=1, argv=0x7fffffffda58) at civserver.c:453
(gdb) c
Continuing.

Game saved as freeciv-T0009-Y-3550-auto.sav.bz2
>
Breakpoint 2, add_clause (ptreaty=0x8064b9ee0, pfrom=0x8063dd400, type=CLAUSE_EMBASSY, val=0)
    at diptreaty.c:138
(gdb)
```

Let's go line by line using next. You could also use the step command but if you use the command, it will take you inside a function call instead of just evaluating the function and returning like the next command.



```
+--diptreaty.c-----------------------------------------------------------------------+
B+ |138                       ? ptreaty->plr1 : ptreaty->plr0);
   |139        struct Clause *pclause;
   |140        enum diplstate_type ds
   |141          = player_diplstate_get(ptreaty->plr0, ptreaty->plr1)->type;
   |142
  >|143        if (type < 0 || type >= CLAUSE_LAST) {
   |144          log_error("Illegal clause type encountered.");
   |145          return FALSE;
   |146        }
   |147
   |148        if (type == CLAUSE_ADVANCE && !valid_advance_by_number(val)) {
   |149          log_error("Illegal tech value %i in clause.", val);
   |150          return FALSE;
   |151        }
   |152
   |153        if (is_pact_clause(type)
   |154            && ((ds == DS_PEACE && type == CLAUSE_PEACE)
   |155                || (ds == DS_ARMISTICE && type == CLAUSE_PEACE)
   |156                || (ds == DS_ALLIANCE && type == CLAUSE_ALLIANCE)
   |157                || (ds == DS_CEASEFIRE && type == CLAUSE_CEASEFIRE))) {
+------------------------------------------------------------------------------------+
freebsd-th Thread 8054064 In: add_clause                         Line: 143   PC: 0x800cf9484
(gdb) f 0
#0  add_clause (ptreaty=0x8064b9ee0, pfrom=0x8063dd400, type=CLAUSE_EMBASSY, val=0) at diptreaty.c:141
(gdb) list
(gdb) n
(gdb) list
(gdb) p type
$14 = CLAUSE_EMBASSY
(gdb) ptype CLAUSE_EMBASSY
type = enum clause_type {CLAUSE_ADVANCE, CLAUSE_GOLD, CLAUSE_MAP, CLAUSE_SEAMAP, CLAUSE_CITY,
    CLAUSE_CEASEFIRE, CLAUSE_PEACE, CLAUSE_ALLIANCE, CLAUSE_VISION, CLAUSE_EMBASSY, CLAUSE_LAST}
(gdb)
```

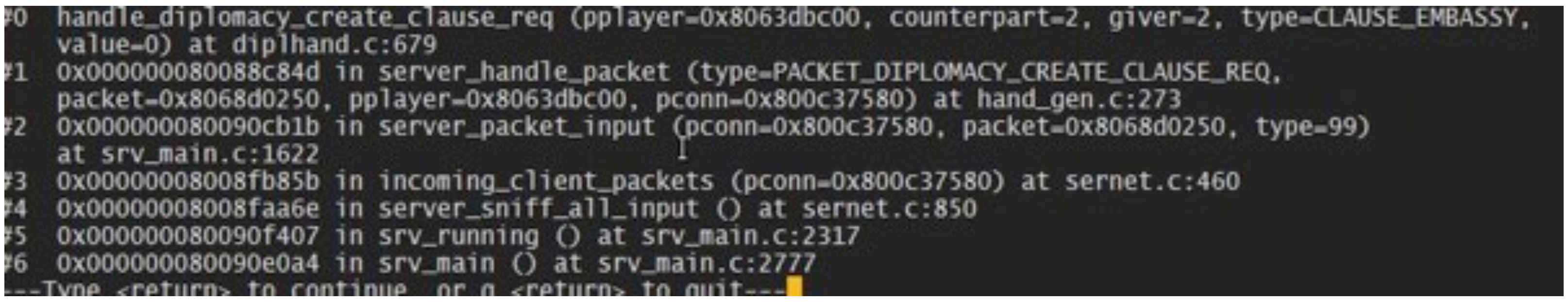


```
+--diptreaty.c-----------------------------------------------------------------------
   |161                     nation_rule_name(nation_of_player(ptreaty->plr0)),
   |162                     nation_rule_name(nation_of_player(ptreaty->plr1)));
   |163          return FALSE;
   |164        }
   |165
  >|166        if (type == CLAUSE_EMBASSY && player_has_real_embassy(pto, pfrom)) {
   |167          /* we already have embassy */
   |168          log_error("Illegal embassy clause: %s already have embassy with %s.",
   |169                    nation_rule_name(nation_of_player(pto)),
   |170                    nation_rule_name(nation_of_player(pfrom)));
   |171          return FALSE;
   |172        }
   |173
   |174        if (!game.info.trading_gold && type == CLAUSE_GOLD) {
   |175          return FALSE;
   |176        }
   |177        if (!game.info.trading_tech && type == CLAUSE_ADVANCE) {
   |178          return FALSE;
   |179        }
   |180        if (!game.info.trading_city && type == CLAUSE_CITY) {
```

We are currently at line 143. I just checked what kind of data type was CLAUSE_EMBASSY. It was an enum one (somewhat obvious).

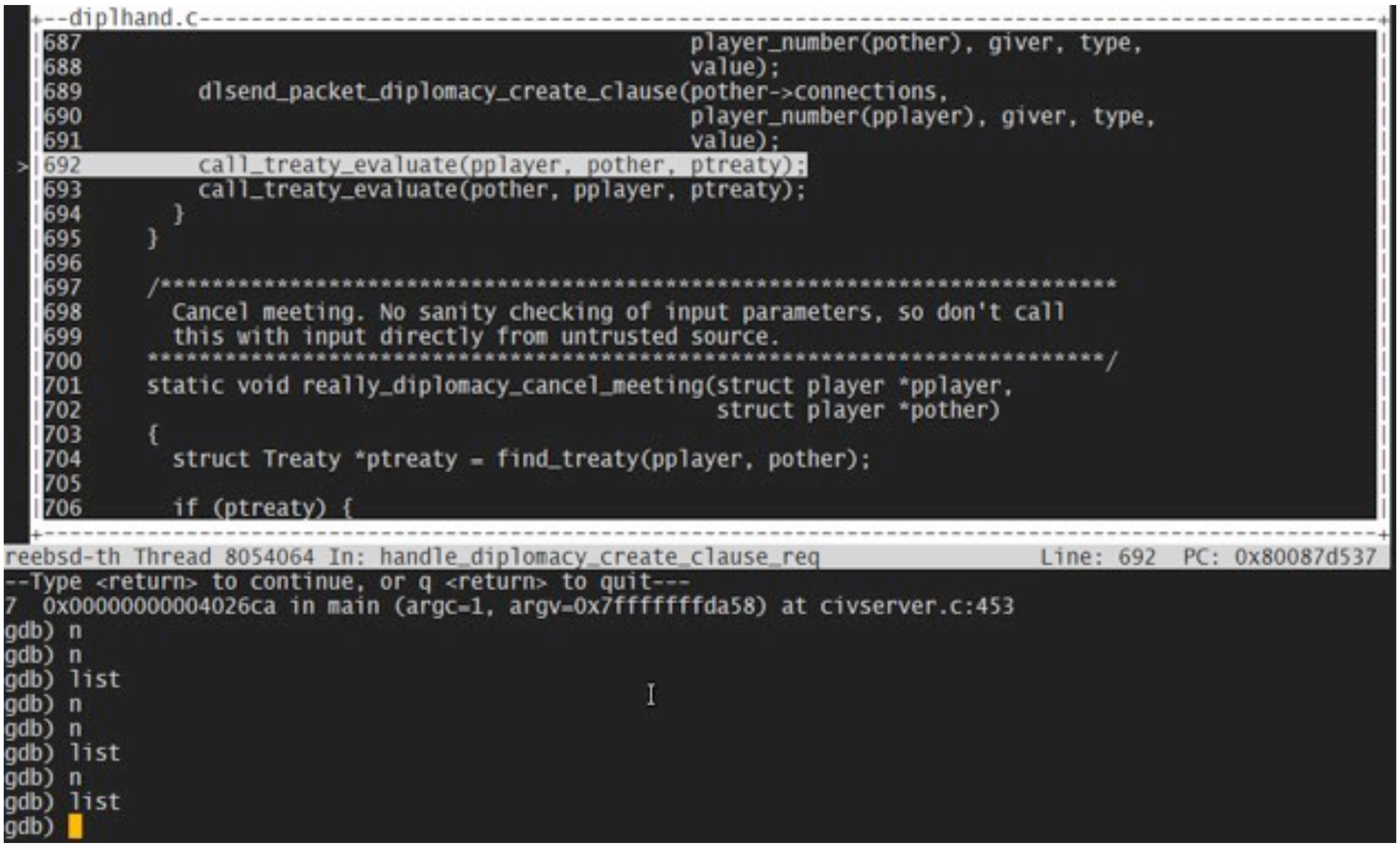
Using ‘next’ command a couple of times will get us here:

Keep on typing ‘n’ and we will exit from the function call and arrive to handle_diplomacy_create_clause_req

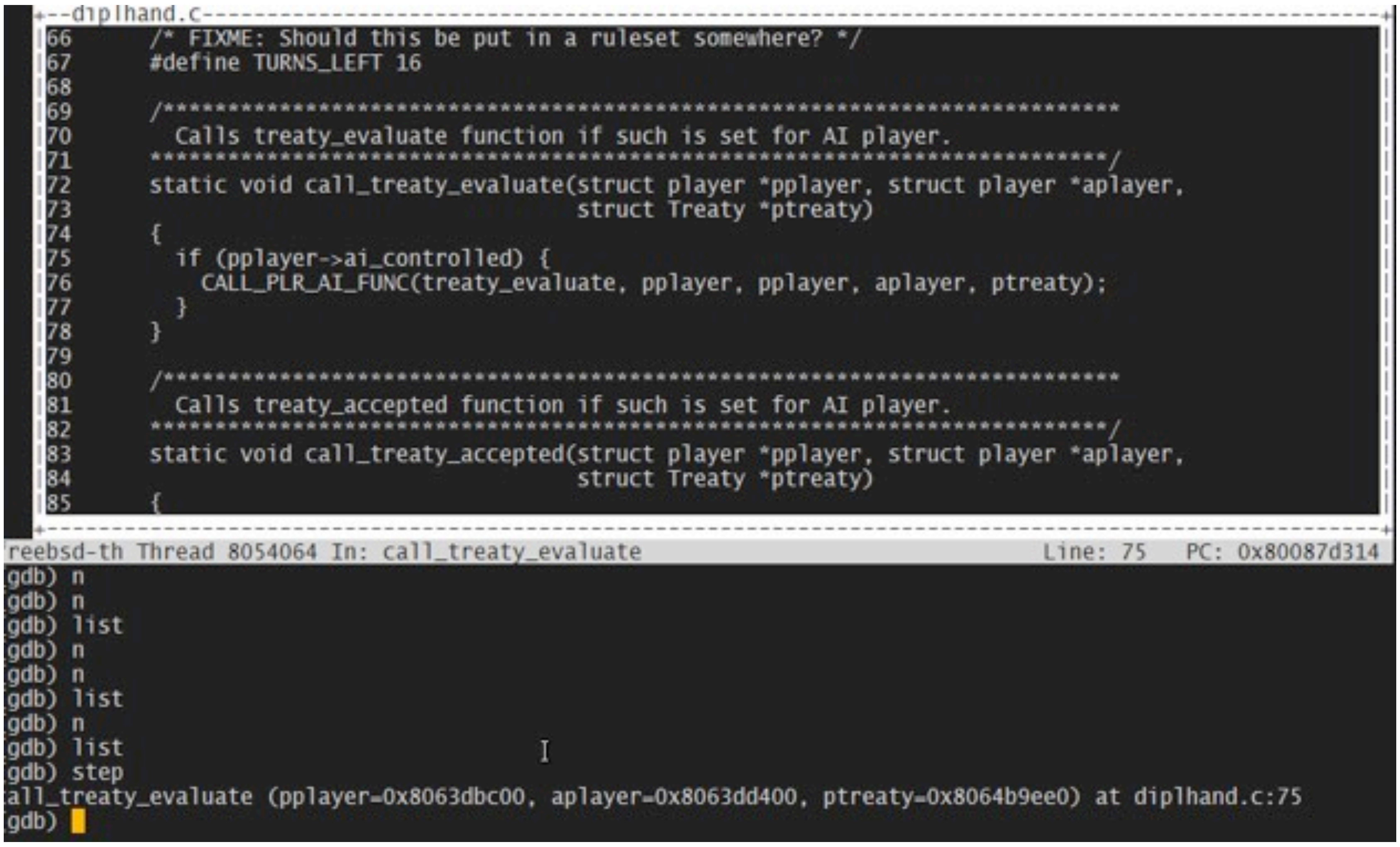


```
#0  handle_diplomacy_create_clause_req (pplayer=0x8063dbc00, counterpart=2, giver=2, type=CLAUSE_EMBASSY,
    value=0) at diplhand.c:679
#1  0x000000080088c84d in server_handle_packet (type=PACKET_DIPLOMACY_CREATE_CLAUSE_REQ,
    packet=0x8068d0250, pplayer=0x8063dbc00, pconn=0x800c37580) at hand_gen.c:273
#2  0x000000080090cb1b in server_packet_input (pconn=0x800c37580, packet=0x8068d0250, type=99)
    at srv_main.c:1622
#3  0x00000008008fb85b in incoming_client_packets (pconn=0x800c37580) at sernet.c:460
#4  0x00000008008faa6e in server_sniff_all_input () at sernet.c:850
#5  0x000000080090f407 in srv_running () at srv_main.c:2317
#6  0x000000080090e0a4 in srv_main () at srv_main.c:2777
---Type <return> to continue, or q <return> to quit---
```

As we keep on typing ‘next’ , we will arrive at this function call_treaty_evaluate that seems interesting. Maybe here the results of rejection or acceptance of conditions are done. As I explained earlier, we can step into this one using the step command.

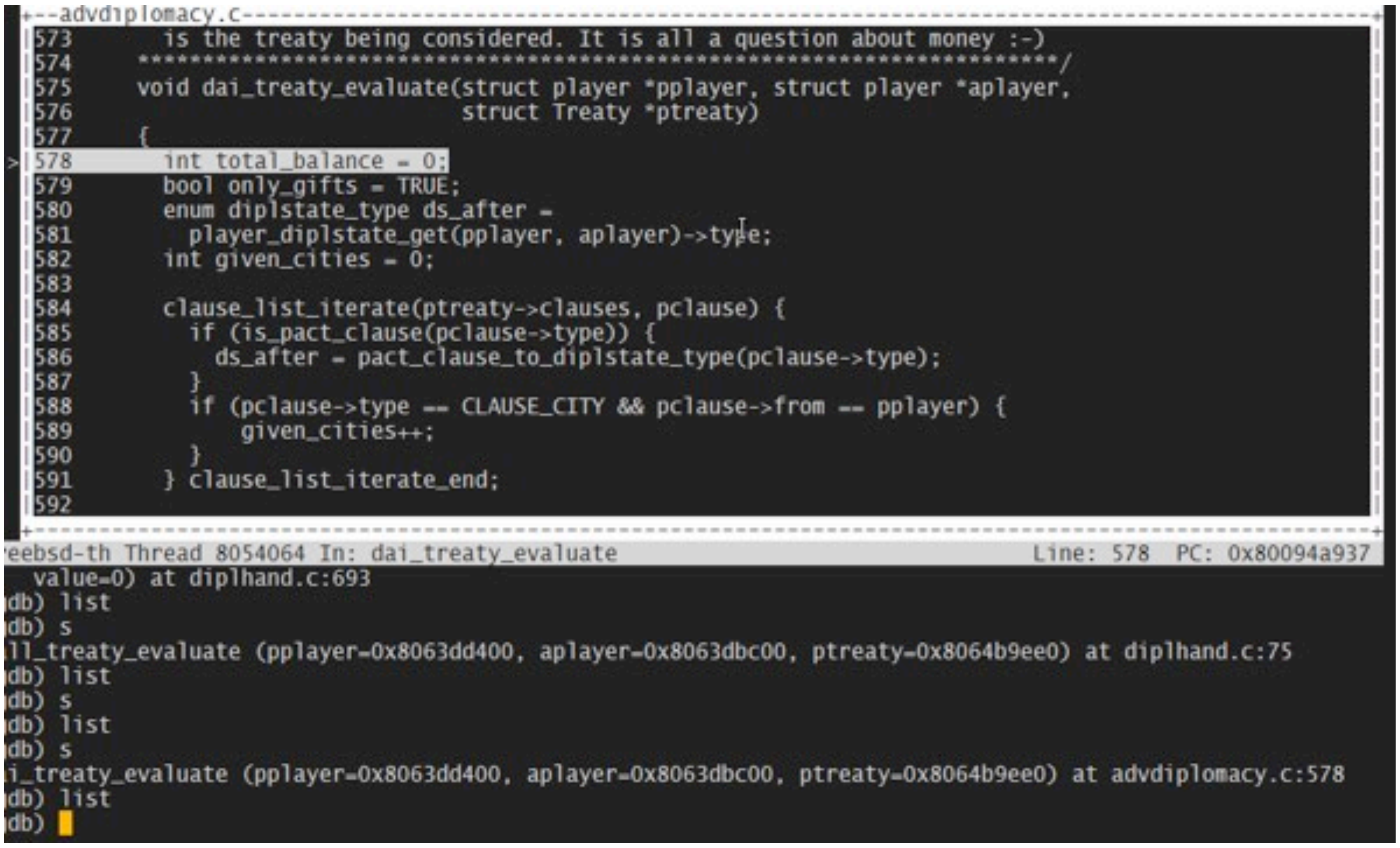


```
+--diplhand.c---------------------------------------------------------------------+
                                         player_number(pother), giver, type,
                                         value);
         dlsend_packet_diplomacy_create_clause(pother->connections,
                                         player_number(pplayer), giver, type,
                                         value);
>        call_treaty_evaluate(pplayer, pother, ptreaty);
         call_treaty_evaluate(pother, pplayer, ptreaty);
       }
     }

     /**************************************************************************
       Cancel meeting. No sanity checking of input parameters, so don't call
       this with input directly from untrusted source.
     **************************************************************************/
     static void really_diplomacy_cancel_meeting(struct player *pplayer,
                                                 struct player *pother)
     {
       struct Treaty *ptreaty = find_treaty(pplayer, pother);

       if (ptreaty) {
+---------------------------------------------------------------------------------+
reebsd-th Thread 8054064 In: handle_diplomacy_create_clause_req      Line: 692   PC: 0x80087d537
--Type <return> to continue, or q <return> to quit---
7  0x00000000004026ca in main (argc=1, argv=0x7fffffffda58) at civserver.c:453
gdb) n
gdb) n
gdb) list
gdb) n
gdb) n
gdb) list
gdb) n
gdb) list
gdb)
```



```
+--diplhand.c---------------------------------------------------------------------+
     /* FIXME: Should this be put in a ruleset somewhere? */
     #define TURNS_LEFT 16

     /**************************************************************************
       Calls treaty_evaluate function if such is set for AI player.
     **************************************************************************/
     static void call_treaty_evaluate(struct player *pplayer, struct player *aplayer,
                                      struct Treaty *ptreaty)
     {
       if (pplayer->ai_controlled) {
         CALL_PLR_AI_FUNC(treaty_evaluate, pplayer, pplayer, aplayer, ptreaty);
       }
     }

     /**************************************************************************
       Calls treaty_accepted function if such is set for AI player.
     **************************************************************************/
     static void call_treaty_accepted(struct player *pplayer, struct player *aplayer,
                                      struct Treaty *ptreaty)
     {
+---------------------------------------------------------------------------------+
reebsd-th Thread 8054064 In: call_treaty_evaluate                    Line: 75   PC: 0x80087d314
gdb) n
gdb) n
gdb) list
gdb) n
gdb) n
gdb) list
gdb) n
gdb) list
gdb) step
all_treaty_evaluate (pplayer=0x8063dbc00, aplayer=0x8063dd400, ptreaty=0x8064b9ee0) at diplhand.c:75
gdb)
```

Let's step all the way to get to another point in the program execution. After a couple of steps, we get to this point:

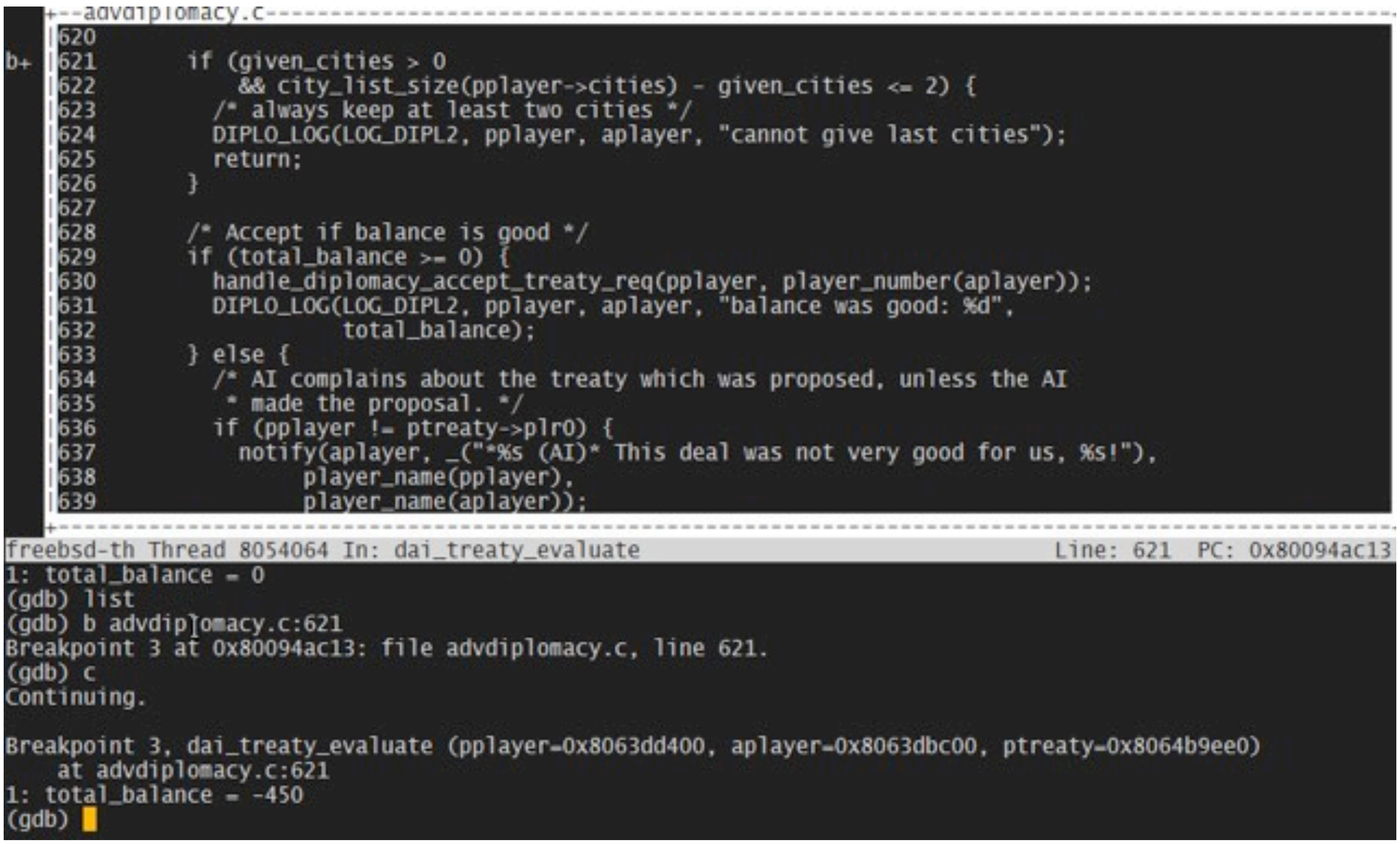


```
+--advdiplomacy.c-----------------------------------------------------------------+
|573       is the treaty being considered. It is all a question about money :-)
|574     ***********************************************************************/
|575     void dai_treaty_evaluate(struct player *pplayer, struct player *aplayer,
|576                              struct Treaty *ptreaty)
|577     {
>|578       int total_balance = 0;
|579       bool only_gifts = TRUE;
|580       enum diplstate_type ds_after =
|581         player_diplstate_get(pplayer, aplayer)->type;
|582       int given_cities = 0;
|583
|584       clause_list_iterate(ptreaty->clauses, pclause) {
|585         if (is_pact_clause(pclause->type)) {
|586           ds_after = pact_clause_to_diplstate_type(pclause->type);
|587         }
|588         if (pclause->type == CLAUSE_CITY && pclause->from == pplayer) {
|589             given_cities++;
|590         }
|591       } clause_list_iterate_end;
|592
+---------------------------------------------------------------------------------+
eebsd-th Thread 8054064 In: dai_treaty_evaluate                 Line: 578  PC: 0x80094a937
  value=0) at diplhand.c:693
db) list
db) s
ll_treaty_evaluate (pplayer=0x8063dd400, aplayer=0x8063dbc00, ptreaty=0x8064b9ee0) at diplhand.c:75
db) list
db) s
db) list
db) s
i_treaty_evaluate (pplayer=0x8063dd400, aplayer=0x8063dbc00, ptreaty=0x8064b9ee0) at advdiplomacy.c:578
db) list
db)
```

So, a quick glance at the source code tells us that the **total_balance** variable is somewhat important to evaluate if a clause is accepted (In our case, we are requesting to give us an embassy). Instead of printing this variable multiple times, let's leave it available on the display.

```
#(gdb) display total_balance
```

Then, we set a breakpoint somewhere ahead of advdiplomacy.c:621. We can see that the **total_balance** value is displayed and it is -450, seems bad for our proposal.



```
+--advdiplomacy.c-----------------------------------------------------------------------+
   |620
b+ |621       if (given_cities > 0
   |622           && city_list_size(pplayer->cities) - given_cities <= 2) {
   |623         /* always keep at least two cities */
   |624         DIPLO_LOG(LOG_DIPL2, pplayer, aplayer, "cannot give last cities");
   |625         return;
   |626       }
   |627
   |628       /* Accept if balance is good */
   |629       if (total_balance >= 0) {
   |630         handle_diplomacy_accept_treaty_req(pplayer, player_number(aplayer));
   |631         DIPLO_LOG(LOG_DIPL2, pplayer, aplayer, "balance was good: %d",
   |632                   total_balance);
   |633       } else {
   |634         /* AI complains about the treaty which was proposed, unless the AI
   |635          * made the proposal. */
   |636         if (pplayer != ptreaty->plr0) {
   |637           notify(aplayer, _("*%s (AI)* This deal was not very good for us, %s!"),
   |638                  player_name(pplayer),
   |639                  player_name(aplayer));
+---------------------------------------------------------------------------------------+
freebsd-th Thread 8054064 In: dai_treaty_evaluate                  Line: 621   PC: 0x80094ac13
1: total_balance = 0
(gdb) list
(gdb) b advdiplomacy.c:621
Breakpoint 3 at 0x80094ac13: file advdiplomacy.c, line 621.
(gdb) c
Continuing.

Breakpoint 3, dai_treaty_evaluate (pplayer=0x8063dd400, aplayer=0x8063dbc00, ptreaty=0x8064b9ee0)
    at advdiplomacy.c:621
1: total_balance = -450
(gdb) 
```

As we can see, **total_balance >=0 is the condition to** approve the proposal. The following is a resume of the commands used in this session:

COMMAND	ABBREVIATED FORM	WHAT IT DOES
info local	**i lo**	*Prints values and names of all local variables in the current scope.*
backtrace	**bt**	*A backtrace is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the current executing frame (frame zero), followed by its caller (frame one), and on up the stack.*
frame <frame number>	**f <frame number>**	*The call stack is divided up into contiguous pieces called stack frames, or frames for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.*
print <variable>	**p <variable>**	*displays the value of the variable*
display <variable>	**disp <variable>**	*Will automatically print the value of the variable being displayed as long as it is within the scope*
win	**Win**	*Will enter gdb in tui (text user interface) mode if we had not entered in the first place. Default layout is source at the top commands at the bottom.*
next	**n** **n <number of next to perform>**	*Executes next line of code. Will not enter functions. You can use as parameter the number or times to execute next*
step	**s** **s <number of steps to perform>**	*Step to next line of code. Will step into a function.*

These are really basic commands, but useful. Armed with these commands, you should be able to answer a couple of questions using gdb and not just browsing the source code:

Describe the flow when a city is created, which are the stack frames?

What is the name of the structure which handles units?

What is the name of the structure that handles game information? What are the contents of this structure at the end of a game?

Advanced Inspection of Data Structures and Variables

Automating Information Display

Now that we have used the display command or the print command, it is getting pretty tedious to inspect a variable or data structure manually by typing p or display every time we hit a breakpoint we have set. There is a command called commands to save us from all this typing.

First, we set a breakpoint where we want to inspect data automatically. In this case, I'll check one of the city functions.

```
(gdb) b city.c:2352

(gdb) 4  breakpoint   keep y   0x0000000800cf1b7b in citizen_base_mood at
city.c:2352
```

Now, we can type the following: commands <breakpoint number>

```
(gdb) commands 4

Type commands  when breakpoint 4 is hit, one per line. Then end with a line
saying  "end".

>
```

After you have set the instructions to be executed after the breakpoint is hit, you could modify them or just erase them like this:

```
(gdb) commands 4

Type commands  when breakpoint 4 is hit, one per line. End with a line saying
"end".

> end
```

Now, if you want to execute something:

```
(gdb)    commands 4
```

Type commands when breakpoint 4 is hit, one per line. End with a line saying "end".

```
> printf "Setting city mood for leader: %s", pplayer->name

> end
```

We can type all the instructions we want to be executed when this breakpoint is hit. Usually, we used ***print*** to display values, but there is a more powerful function called ***printf*** that uses similar format as the C-language function

```
(gdb)    printf   "%s", pplayer->name
```

As in C's printf, ordinary characters in the template are printed verbatim, while conversion specification introduced by the '%' character causes subsequent expressions to be evaluated, their values converted and formatted according to type and style information encoded in the conversion specifications, and then printed.

For example, you can print two values in hex like this: printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo

printf supports all the standard C conversion specifications, including the flags and modifiers between the '%' character and the conversion letter, with the following exceptions:

- The argument-ordering modifiers, such as '2$', are not supported. The modifier '*' is not supported for specifying precision or width.
- The ''' flag (for separation of digits into groups according to LC_NUMERIC') is not supported. The type modifiers 'hh', 'j', 't', and 'z' are not supported.
- The conversion letter 'n' (as in '%n') is not supported. The conversion letters 'a' and 'A' are not supported.

Note that the 'll' type modifier is supported only if the underlying C implementation used to build GDB supports the long long int type. Likewise, the 'L' type modifier is supported only if long double type is available.

As in C, printf supports simple backslash-escape sequences such as \n, '\t', '\\', '\"', '\a', and '\f', that consist of backslash followed by a single character. Octal and hexadecimal escape sequences are not supported.

Additionally, printf supports conversion specifications for DFP (Decimal Floating Point) types using the following length modifiers together with a floating point specifier. letters:

'H' for printing Decimal32 types. 'D' for printing Decimal64 types.

'DD' for printing Decimal128 types.

If the underlying C implementation used to build GDB supports the three length modifiers for DFP types, other modifiers such as width and precision will also be available for GDB to use.

In case there is no such C support, no additional modifiers will be available, and the value will be printed in the standard way.

Here’s an example of printing DFP types using the above conversion letters: printf "D32: %Hf - D64: %Df - D128: %DDf\n",1.2345df,1.2E10dd,1.2E1dl

Dynamically allocated arrays

Sometimes we will need to take a look at the contents of dynamically allocated arrays (the ones created by *malloc* and *calloc* system calls).

For example, we have the usual static memory array:

```
char t[8001];
```

It’s easy to display its contents using:

```
(gdb) p t
```

But what about this one:

```
int *t; ...
```

```
t = (int *) malloc ( 8001 * sizeof( int) ); (gdb) p t
```

This will give only the address:

```
(gdb) p *t
```

This will give you the data of the first element in the array, so what is the solution?

```
(gdb) p *t@25
```

This command will print 25 elements from the array t .The format is pointer@<number of elements.

Getting information from the symbol table

When we compiled our program with the –g flag, we instructed the compiler to generate a symbol table in our program binary. The table contains variable names, function names and types. Now, let’s suppose we want to know the names of all the functions available, we could use one of the info family commands:

```
(gdb)    info functions
```

This command will print the names and data types of all defined functions. If we want to check only the function names matching a *regexp* ,we use the command: info functions <*regexp*>

For example:

```
(gdb)    info functions city
```

The above command matches all functions that have city string in their name. You must use grep regexp not perl’s regexp . The same goes with variables with the command:

```
(gdb)    info variables
```

Prints the names and data types of all variables that are declared outside of functions (not the local variables).

Also, the same syntax for info variables *regexp*

```
(gdb) info variables city
```

Prints the names and data types of all variables (except for local variables) whose names contain a match for regular expression *regexp*.

```
(gdb) info address symbol
```

Describes where the data for symbol is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored. Note the contrast with `print &symbol', which does not work at all for a register variable. For a stack local variable, it prints the exact address of the current instantiation of the variable.

```
(gdb) whatis exp
```

Prints the data type of expression exp. exp is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. Any kind of constant, variable or operator defined by the programming language you are using is valid in an expression in GDB.

```
(gdb) whatis
```

Prints the data type of $, the last value in the value history.

```
(gdb) ptype typename
```

Prints a description of data type typename. typename may be the name of a type, or for C code it may have the form `class class-name', `struct struct-tag', `union union-tag' or `enum enum-tag'.

```
(gdb) ptype   exp ptype
```

Prints a description of the type of expression exp. ptype differs from whatis by printing a detailed description, instead of just the name of the type. For example, for this variable declaration:

```
struct example {double dtype; float ftype} ex1;
```

The two commands give this output:

```
(gdb) whatis  ex1 type = struct example (gdb) ptype ex1

type = struct example { double  dtype; float ftype;

}
```

As with whatis, using ptype without an argument refers to the type of $, the last value in the value history.

```
(gdb) info types regexp
```

Prints a brief description of all types whose name matches regexp (or all types in your program, if you supply no argument). Each complete typename is matched as though it were a complete line; thus, `i type value' gives information on all types in your program whose name includes the string value. But `i type ^value$' only gives information on types whose complete name is value. This command differs from ptype in two ways: first, like whatis, it does not print a detailed description; second, it lists all source files where a type is defined.

```
(gdb) info source
```

Shows the name of the current source file--that is, the source file for the function containing the current point of execution--and the language it was written in. **`(gdb) info sources`**

Prints the names of all source files in your program for which there is debugging information, organized into two lists: files whose symbols have already been read, and files whose symbols will be read when needed.

```
(gdb) info functions
```

Prints the names and data types of all defined functions.

```
(gdb) info functions regexp
```

Prints the names and data types of all defined functions whose names contain a match for regular expression regexp. Thus, `info fun step' finds all functions whose names include step;

`info fun ^step' finds those whose names start with step.

```
(gdb) info variables
```

Prints the names and data types of all variables that are declared outside of functions (i.e., excluding local variables).

```
(gdb) info variables regexp
```

Prints the names and data types of all variables (except for local variables) whose names contain a match for regular expression regexp.

Hey! Stop and look around

In GDB, we have three ways of interrupting the program flow and inspect what we need. We have Breakpoints, watchpoints and catch points.

A breakpoint stops the execution at a particular location within the program. We have temporary breakpoints, *regexp* breakpoints and we could set conditional breakpoints.

The usual breakpoint:

```
(gdb) break <source>:<line>
```

```
(gdb) break <source.c>:<function>
```

(gdb) break 3 ← This one stops at line 3 of the current source file being executed.

```
(gdb) break <function>
```

The temporary break point is a simple breakpoint that is deleted after it is hit. The command for this is:

```
(gdb) tbreak <same format as breakpoint>
```

The regexp breakpoint sets breakpoints at the functions matching the regexp provided

```
(gdb) rbreak ^city
```

Conditional breakpoint, stops the execution of the program only if the condition is met.

```
(gdb) b if strcmp(commands[0].synopsis,"*start") ==0
```

Yes, you could use the C library functions as long as your program is linked against libc. You can enable or disable breakpoints with the following command:

enable once -- Enable breakpoints for one hit

enable delete -- Enable breakpoints and deletes when hit

```
(gdb) enable once 1 (gdb) enable delete 1
```

A watchpoint stops the execution when a particular memory location (or an expression involving one or more locations) changes value. Depending on your system, watchpoints may be implemented in software or hardware. GDB does software watchpointing by single-stepping your program and testing the variable's value each time, which is hundreds of times slower than normal execution. However, it’s really useful if you don’t have a clue of where the problem is in your program.

The syntax for this command is: watch <expr>

```
(gdb) watch commands[0] Watchpoint 1: commands[0]
```

A catchpoint stops the execution when a particular event occurs. The event could be one of the following:

Raised signals may be caught:

catch signal - all signals

catch signal <signame> - a particular signal

Raised exceptions may be caught:

catch throw- all exceptions, when thrown

catch throw <exceptname> - a particular exception, when thrown catch catch - all exceptions, when caught

catch catch <exceptname> - a particular exception, when caught

Thread or process events may be caught:

catch thread_start - any threads, just after creation catch thread_exit - any threads, just before expiration catch thread_join- any threads, just after joins

Process events may be caught:

catch start - any processes, just after creation catch exit - any processes, just before expiration catch fork - calls to fork()

catch vfork - calls to vfork()

catch exec - calls to exec()

Dynamically-linked library events may be caught:

catch load - loads of any library

catch load <libname> - loads of a particular library catch unload - unloads of any library

catch unload <libname> - unloads of a particular library

The act of your program's execution stopping may also be caught:

catch stop

C++ exceptions may be caught:

catch throw - all exceptions, when thrown catch catch - all exceptions, when caught

You can enable and delete breakpoints, watchpoints and catchpoints with the enable and delete command.

Proposed Exercises:

Set a breakpoint that gets triggered when the size of a city is greater than 20 citizens.

Do you know what functions modify the game data type? Find out using a watchpoint.

Display leader’s and city names when a city’s data type is being read or modified by the program.

When is a government during a revolution being set? Use a watchpoint or a conditional breakpoint.

For any questions or issues, just use the forum
http://bsdmag.org/forums/forum/application-debugging-and-troubleshooting/

Have fun.

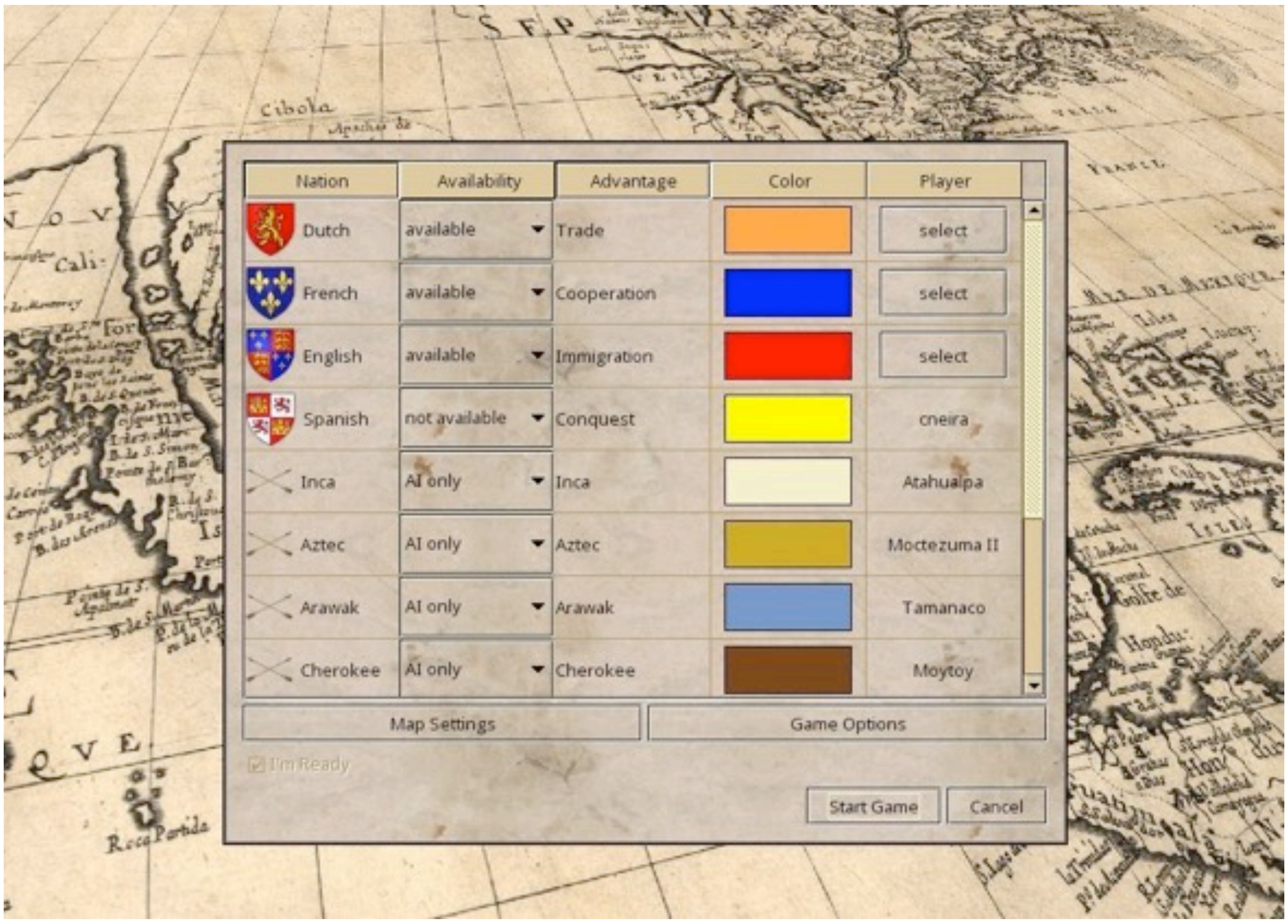
Introduction to The JDB

The Java Debugger

“The Java Debugger (JDB) is a simple command-line debugger for Java classes. The jdb command and its options call the JDB. The jdb command demonstrates the Java Platform Debugger Architecture (JDBA) and provides inspection and debugging of a local or remote Java Virtual Machine (JVM). See Java Platform Debugger Architecture (JDBA) at http://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html”

JDB, like GDB, is a command line debug tool that will help us find bugs in our code (it is provided by openjdk), hopefully with little effort. The application that will be used to demonstrate the usefulness of JDB will follow the setting as the last one. The application we are going to use is called Freecol:

"The FreeCol team aims to create an Open-Source version of Colonization (released under the GPL). At first, we'll try to make an exact clone of Colonization. "



Nation	Availability	Advantage	Color	Player
Dutch	available	Trade	🟧	select
French	available	Cooperation	🟦	select
English	available	Immigration	🟥	select
Spanish	not available	Conquest	🟨	cneira
Inca	AI only	Inca	⬜	Atahualpa
Aztec	AI only	Aztec	🟨	Moctezuma II
Arawak	AI only	Arawak	🟦	Tamanaco
Cherokee	AI only	Cherokee	🟫	Moytoy

Like the last time we used GDB, we need the Freecol application sources to be compiled with extra debugging symbols (-g flag to the java compiler).

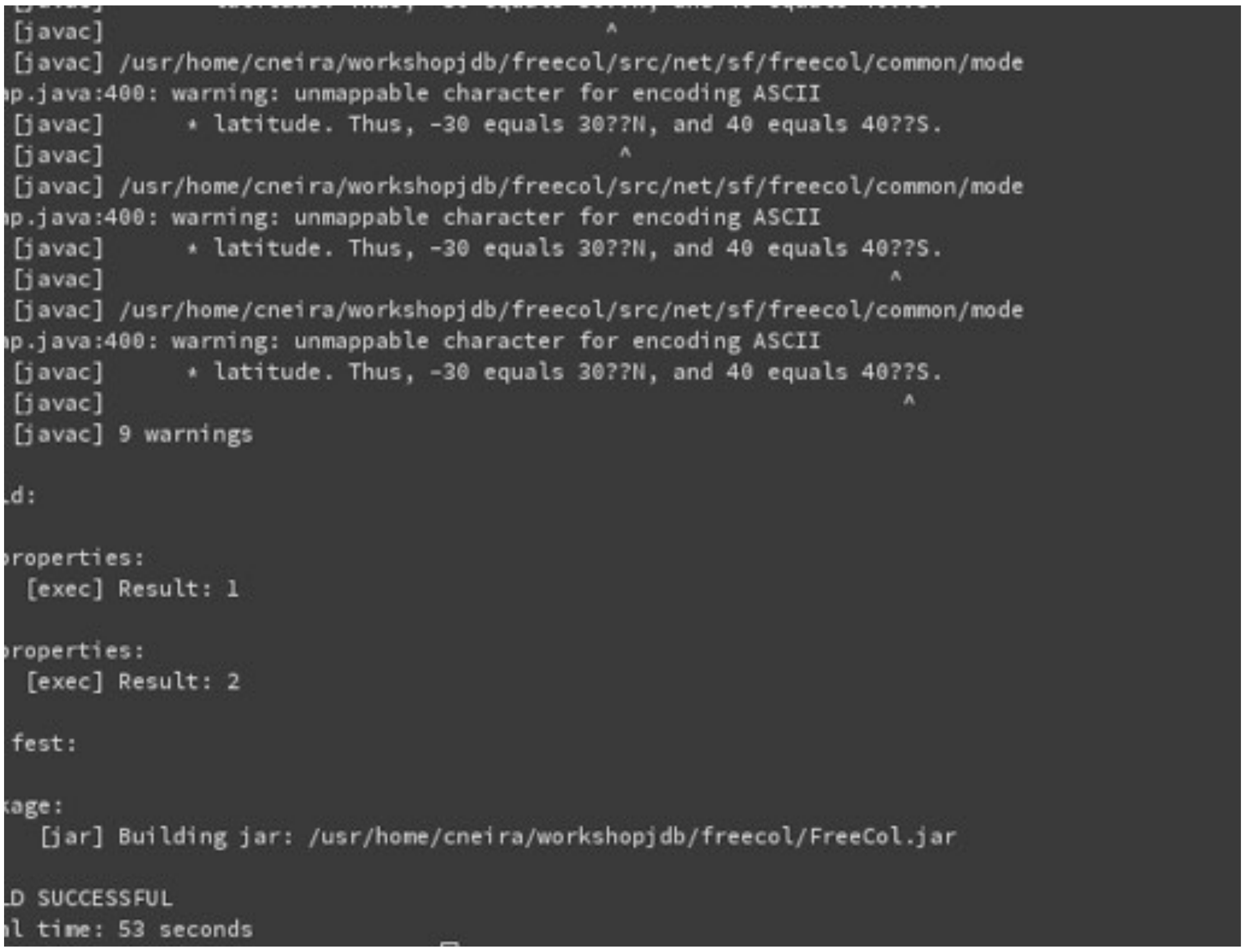
Requirements: You must download the following package :
http://prdownloads.sourceforge.net/freecol/freecol- 0.10.7-src.zip?download

You must have the openjdk7 package installed 3- You must have apache-ant installed

Initial Steps: Go to the folder where you have extracted the Freecol sources

Type ‘ant’. This will start building the Freecol application with debugging information (you will see as was in GDB, the -g flag is used by the compiler if you look at the build.xml).

This will take some minutes and you will have a new FreeCol.jar file



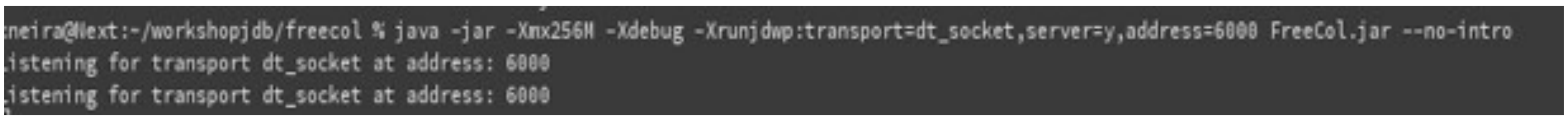
```
[javac]                                                      ^
[javac] /usr/home/cneira/workshopjdb/freecol/src/net/sf/freecol/common/mode
p.java:400: warning: unmappable character for encoding ASCII
[javac]      * latitude. Thus, -30 equals 30??N, and 40 equals 40??S.
[javac]                                                       ^
[javac] /usr/home/cneira/workshopjdb/freecol/src/net/sf/freecol/common/mode
p.java:400: warning: unmappable character for encoding ASCII
[javac]      * latitude. Thus, -30 equals 30??N, and 40 equals 40??S.
[javac]                                                                  ^
[javac] /usr/home/cneira/workshopjdb/freecol/src/net/sf/freecol/common/mode
p.java:400: warning: unmappable character for encoding ASCII
[javac]      * latitude. Thus, -30 equals 30??N, and 40 equals 40??S.
[javac]                                                                   ^
[javac] 9 warnings

d:

roperties:
  [exec] Result: 1

roperties:
  [exec] Result: 2

 fest:

age:
   [jar] Building jar: /usr/home/cneira/workshopjdb/freecol/FreeCol.jar

D SUCCESSFUL
l time: 53 seconds
```

Start debugging

There are two ways to start debugging using JDB.

The first one is to give JDB the initial class (the one that has the main function) and start from there. As the JVM has not started, you must type ‘run’ to start the program. We will use the second approach that is to start a JVM and connect to it:

$ java -jar -Xmx256M -Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=6000 FreeCol.jar --no-intro



```
neira@Next:~/workshopjdb/freecol % java -jar -Xmx256M -Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=6000 FreeCol.jar --no-intro
istening for transport dt_socket at address: 6000
istening for transport dt_socket at address: 6000
```

Here, I use the --no-intro parameter because there is an issue with Freecol where the intro movie freezes the application.

Then, we need to attach to this running JVM with the following command:

`$ jdb -attach    localhost:6000`

Now, let's start with some basic commands.

Setting breakpoints :

First, we need to know how to set breakpoints:

`stop in <class-name>.<method-name>` *Stop on entry to the given method.*

`stop at <class-name>:<line-number>` *Stop at the given line.*

`clear <class-name>.<method-name>` *Remove the specified breakpoint.*

`clear <class-name>:<line-number>` *Remove the specified breakpoint.*

Stepping in Stepping out

When you hit a breakpoint, you either take a look at the data or continue executing the program. Here is the syntax for continuing, stepping, etc.

cont

As the name suggests, continue the execution after you hit the breakpoint.

step

Execute the current line . If the breakpoint is at a method call, it will stop the program at the method entry (same as GDB).

next

Execute the current line. If the breakpoint is at a method call, do not stop at the method entry (same as GDB).

step up

Execute until the current method returns to its caller (in GDB, the command is called finish).

Take a look at the source code:

Like in GDB, this command bears the same name. You need to execute jdb where the source is located or use the command in the JDB and provide the directory with the source code.

list *Lists 10 lines starting 4 before the current line.*

list <linenumber> *Lists 10 lines starting 4 before the given line.*

list <method_name> *Lists the first 10 lines of the given method.*

Taking a peek

To look at the values of variables or expressions, we have the following commands:

print <name> *Prints the current value of the given variable.*

print <expression> *Prints the value of the given expression.*

locals *Prints the values of all variables local to the current method (In GDB, this one is called Info locals).*

Calling for help

The same as GDB, just type ‘help’.

Option flags

When you use the jdb command instead of the java command on the command line, the jdb command accepts similar options as those accepted by java command, including -D, -classpath, and -X options. The following list contains additional options that are accepted by the jdb command.

-help

Displays a help message.

-sourcepath *dir1:dir2: . . .*

Uses the specified path to search for source files in the specified path. If this option is not specified, then use the default path of dot (.).

-attach *address*

Attaches the debugger to a running JVM with the default connection mechanism.

-listen *address*

Waits for a running JVM to connect to the specified address with a standard connector.

-launch

Starts the debugged application immediately upon startup of JDB. The -launch option removes the need for the run command. The debugged application is launched and then stopped just before the initial application class is loaded. At that point, you can set any necessary breakpoints and use the cont command to continue execution.

-listconnectors

Lists the connectors available in JVM.

-**connect connector-name:*name1=value1***

Connects to the target JVM with the named connector and listed argument values.

-dbgtrace [*flags*]

Prints information for debugging the jdb command.

-tclient

Runs the application in the Java HotSpot VM client.

-tserver

Runs the application in the Java HotSpot VM server.

-J*option*

Passes option to the JVM, where, option is one of the options described on the reference page for the Java application launcher. For example, -J-Xms48m sets the startup memory to 48 MB.

Exercise:
Set a breakpoint when a new Colony is built, and print the stacktrace

Introduction to The JDB

Start Debugging

JDB, like GDB, is a cli-debug tool that will help us find bugs in our code, hopefully with little effort. The application that will be used to demonstrate the usefulness of JDB will follow the setting as the last one. The application we are going to use is called Freecol.
"The FreeCol team aims to create an Open-Source version of Colonization (released under the GPL). At first, we'll try to make an exact clone of Colonization. "
Like the last time we used GDB, we need the Freecol application sources to be compiled with extra debugging symbols (-g flag to the Java compiler).

Requirements :

1- You must download the following package :
http://prdownloads.sourceforge.net/freecol/freecol-0.10.7-src.zip?download
2- You must have the openjdk7 package installed.
3- You must have apache-ant installed

Initial Steps:

1- Go to the folder where you have extracted the Freecol sources.
2- Type 'ant'. This will start building the Freecol application with debugging information (you will see as was in GDB, the -g flag being used by the compiler if you look at the build.xml).
3- This will take some minutes and you will have a new FreeCol.jar file.

Start debugging

There are two ways to start debugging using JDB.
The first one is to give JDB the initial class (the one that has the main function) and start from there. As the JVM has not started, you must type 'run' to start the program. We will use the second approach that is to start a JVM and connect to it:

```
$ java -jar -Xmx256M -Xdebug
-Xrunjdwp:transport=dt_socket,server=y,address=6000 FreeCol.jar --no-intro
```

Here, I use the --no-intro parameter because there is an issue with Freecol where the intro movie freezes the application.

Then, we need to connect it to the running JVM with the following command:

```
$ jdb -attach  localhost:6000
```

Now, let's start with some basic commands.

Setting breakpoints:

First, we need to know how to set breakpoints:

stop in <class-name>.<method-name>	Stop on entry to the given method.
stop at <class-name>:<line-number>	Stop at the given line.
clear <class-name>.<method-name>	Remove the specified breakpoint.
clear <class-name>:<line-number>	Remove the specified breakpoint.

Stepping in Stepping out
When you hit a breakpoint, you either take a look at the data or continue executing the program. Here is the syntax for continuing, steping, etc.:

cont

As the name suggests, continue the execution after you hit the breakpoint.
step

Execute the current line . If the breakpoint is at a method call, it will stop the program at the method entry (same as GDB).
next

Execute the current line. If the breakpoint is at a method call, do not stop at the method entry (same as GDB).
step up

Execute until the current method returns to its caller (In GDB, the command is called finish).

Take a look at the source code:

Like in GDB, this command bears the same name.

list	Lists 10 lines starting 4 before the current line.
list <linenumber>	Lists 10 lines starting 4 before the given line.
list <method_name>	Lists the first 10 lines of the given method.

Taking a peek

To look at the values of variables or expressions, we have the following commands:

print <name>	Prints the current value of the given variable.
print <expression>	Prints the value of the given expression.
locals	Prints the values of all variables local to the current method.

Calling help
The same as GDB, just type ‘help’.
Here is a table of the command syntax between GDB and JDB.

JDB commands	GDB commands
step	s
next	n
cont	c
stop in/at	b
clear	info b
step up	finish
up/down	f
where	bt
print/dump	p

Working with Core Dumps in GDB

What is a core dump?

A core dump or core file is produced when a signal (man signal) indicates that is not possible to continue executing the program because of an error (for example SIGSEV). A core file contains the current state of the program when it was executing (the stack of each thread, contents of the CPU registers, the values of global and static variables, etc..), and if we are lucky and the program that dumped core was compiled with debug options, the core file will have information about the source code and lines. A core file is used to do a postmortem analysis of what happened to our application. If we have the same versions of libraries between hosts, you could take a core produced, let's say on a production machine, and analyze it in another machine that has the same libraries and versions (it is a must. Otherwise, your core file analysis will be invalid).

Allowing to dump cores

For the examples, we are going to use FreeBSD 10.0-STABLE to check if your installation is enabled to generate core files. You could execute the following commands as root:

```
# sysctl -a |grep kern.coredump kern.coredump: 0

# sysctl kern.coredump=1 kern.coredump: 0 -> 1

# sysctl -a | grep kern.coredump kern.coredump: 1
```

kern.coredump = 0 means that the core files will not be generated. Thus, use the sysctl (man sysctl) facilities to change that, and make it permanent by editing /etc/sysctl.conf (man sysctl.conf).

You could see the difference by just executing this simple program:

```
#include<stdio.h>

#include<stdlib.h>

int main() { abort(); }
```

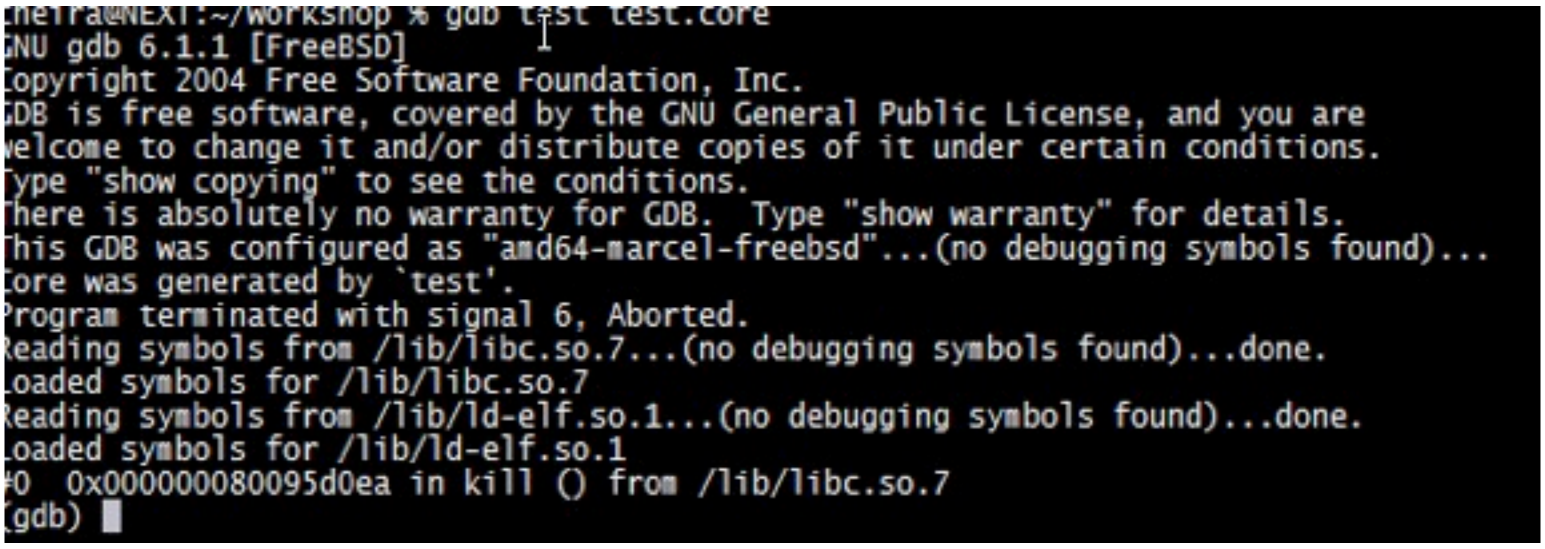
Just name it test.c or whatever name you want. Then, do a make test or make <the name you have chosen> and you will see it core dumping and leaving a test.core file or none if you did not apply the changes to the kern.coredump variable. The abort() causes the program to abnormally terminate hence generating a core file.

Working with the core file

Now that we have a core file, to take a look at it and start debugging, we just type:

```
% gdb test test.core
```

```
% gdb <name of the executable must be in your path> <core file>
```



```
neira@NEXT:~/workshop % gdb test test.core
NU gdb 6.1.1 [FreeBSD]
opyright 2004 Free Software Foundation, Inc.
DB is free software, covered by the GNU General Public License, and you are
elcome to change it and/or distribute copies of it under certain conditions.
ype "show copying" to see the conditions.
here is absolutely no warranty for GDB.  Type "show warranty" for details.
his GDB was configured as "amd64-marcel-freebsd"...(no debugging symbols found)...
ore was generated by `test'.
rogram terminated with signal 6, Aborted.
eading symbols from /lib/libc.so.7...(no debugging symbols found)...done.
oaded symbols for /lib/libc.so.7
eading symbols from /lib/ld-elf.so.1...(no debugging symbols found)...done.
oaded symbols for /lib/ld-elf.so.1
0  0x000000080095d0ea in kill () from /lib/libc.so.7
gdb)
```

As you see, there are no debugging symbols. I used the make command which took my CFLAGS from the

/etc/make.conf file, and there, I did not have the –g flag set.

Debugging without source code

If debugging sometimes with the source code proves cumbersome, definitely, without the source code could be complicated. But let's try debugging without it!

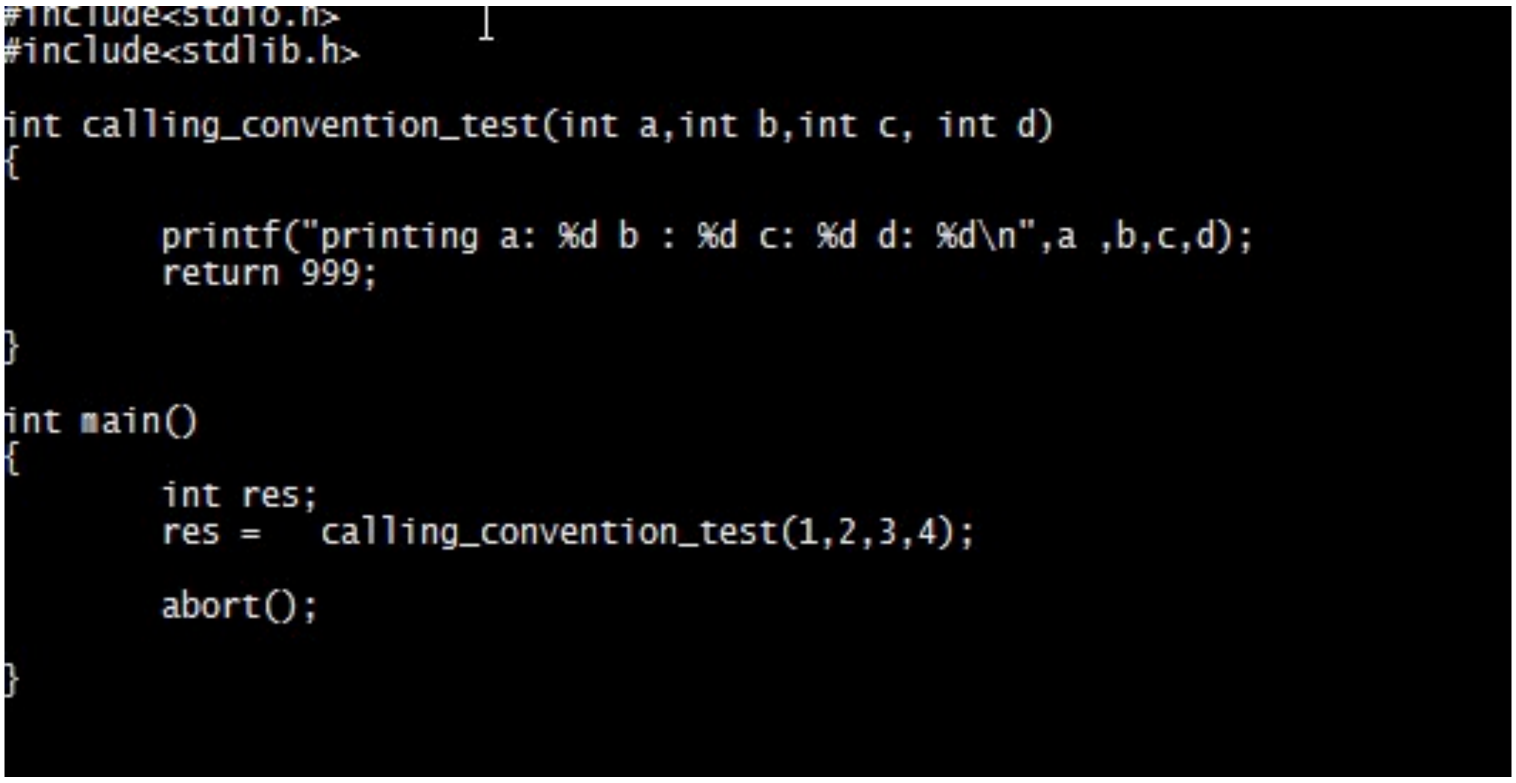
We need to introduce the concept of ***Application Binary Interface***. An **ABI** defines how system calls parameters are passed (which registers contain which argument), how functions are called and in which binary format information should be passed from one program component to another. FreeBSD ABI conforms to these specifications:

http://www.x86-64.org/documentation/abi.pdf

http://www.sco.com/developers/devspecs/abi386-4.pdf

Both documents are a good reading. You could also take a look at the developer's handbook: http://www.nl.freebsd.org/doc/en/books/developers-handbook/x86-system-calls.html

Now, maybe we are in a better position to debug some core files. Let's try this simple program:



```
#include<stdio.h>
#include<stdlib.h>

int calling_convention_test(int a,int b,int c, int d)
{

        printf("printing a: %d b : %d c: %d d: %d\n",a ,b,c,d);
        return 999;

}

int main()
{
        int res;
        res =   calling_convention_test(1,2,3,4);

        abort();

}
```

I'm on x86_64 architecture. So, let's see page 18, section 3.2.3 Parameter passing. In a nutshell, it says that integer parameters (CLASS TYPE INTEGERS) will be passed using registers %rdi,%rsi,%rdx,%rcx,%r8 and %r9. To check that, compile the program by typing:

```
$ cc <source.c> -o <exec name>
```

Then, run the executable program that will get you a core file and load this into GDB as usual.

```
$ gdb <exec> <core file>
```

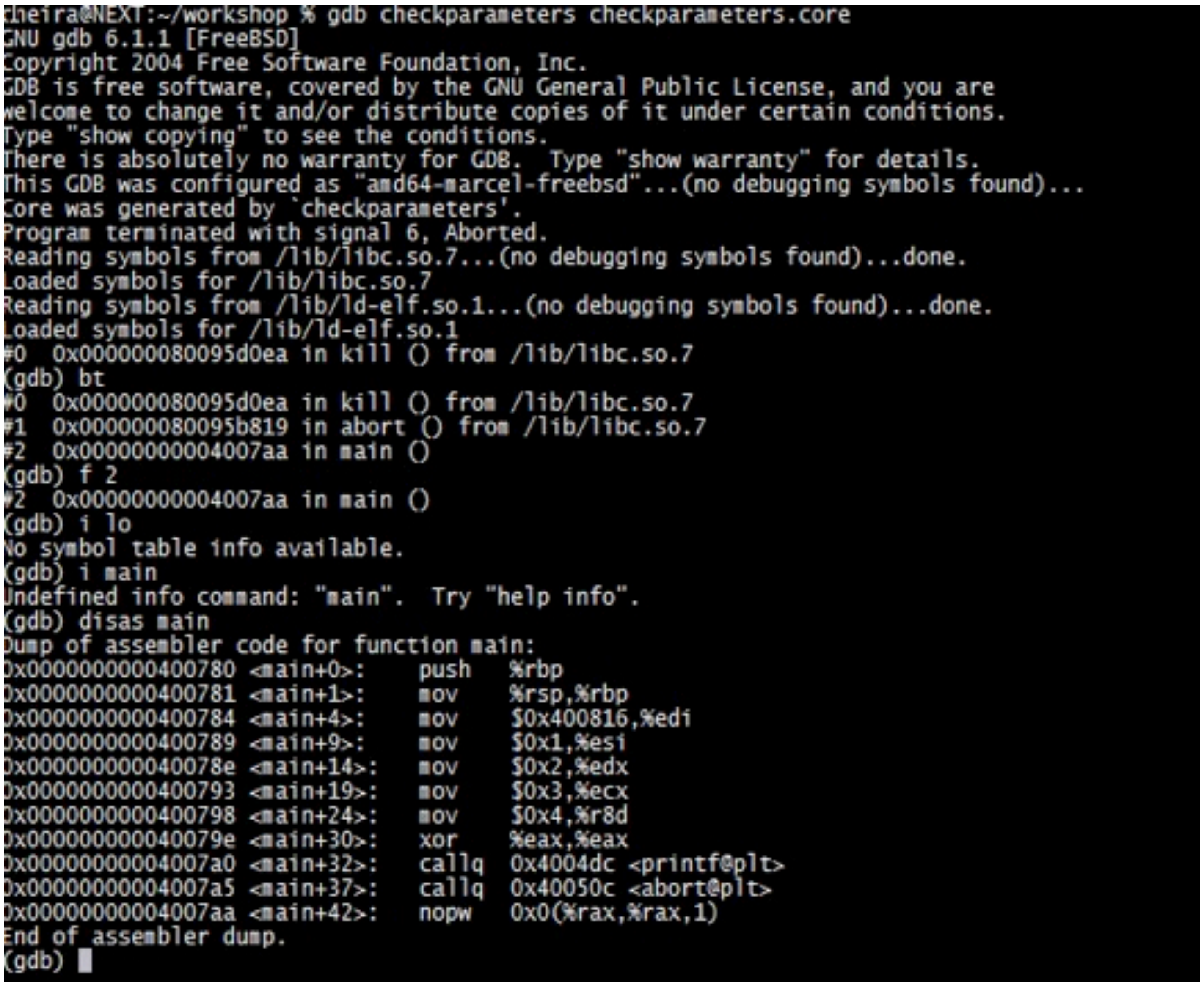
Now, we will use the **disassemble** command (disas is the short form). This command dumps a range of memory as machine instructions. The default memory range is the function surrounding the program counter of the selected frame. Two arguments are interpreted as a range of memory to dump. By default, GDB uses the AT&T notation (http://en.wikipedia.org/wiki/X86_assembly_language). If we want to disassemble a specific function, just type:

```
(gdb) disas <function name>
```

We will always have a main function, but what about the names of the other ones? You could type:

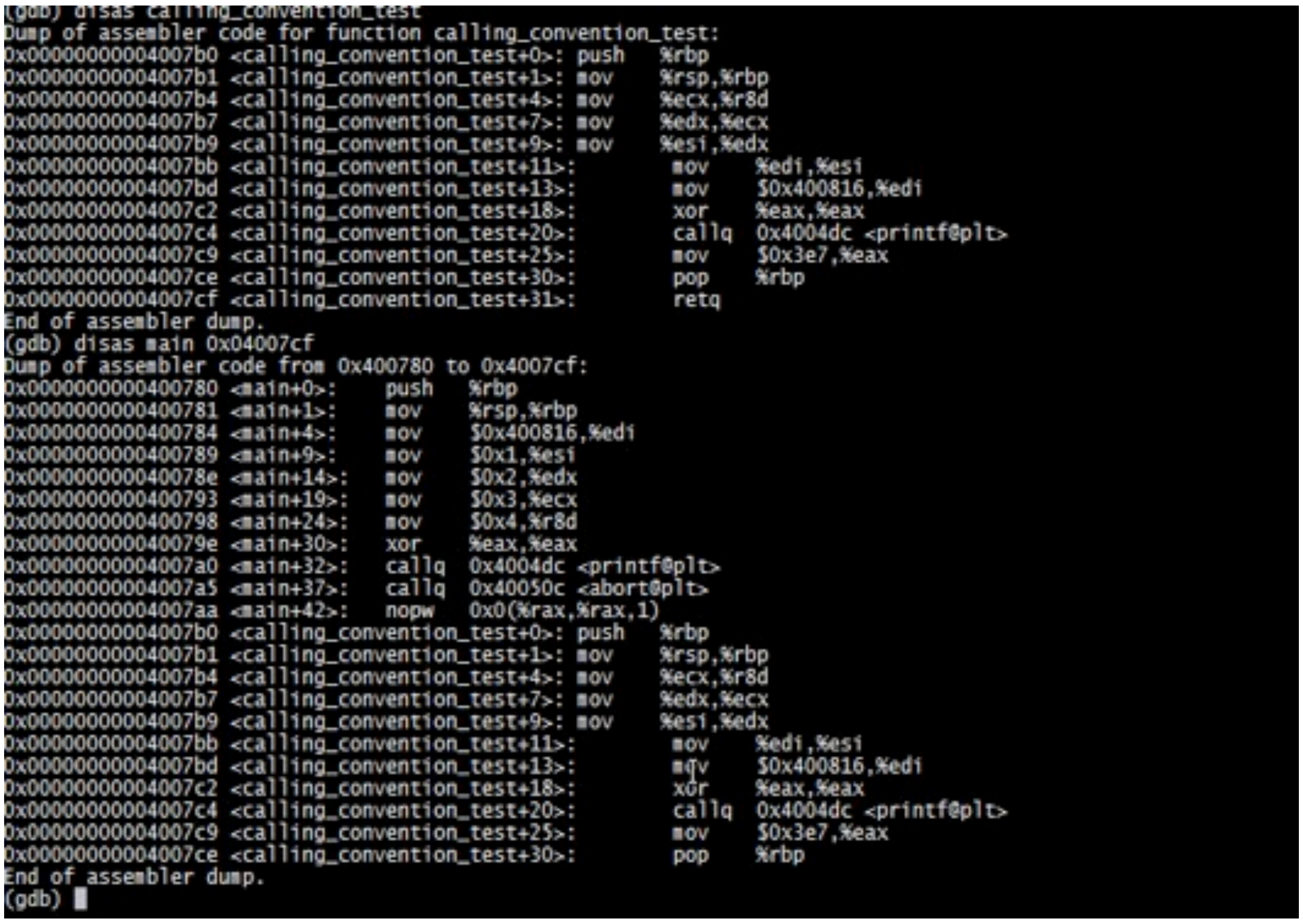
```
(gdb) info func
```

And you will get all function names. This works if the executable has not been stripped (man 1 strip).



```
cheira@NEXT:~/workshop % gdb checkparameters checkparameters.core
GNU gdb 6.1.1 [FreeBSD]
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "amd64-marcel-freebsd"...(no debugging symbols found)...
Core was generated by `checkparameters'.
Program terminated with signal 6, Aborted.
Reading symbols from /lib/libc.so.7...(no debugging symbols found)...done.
Loaded symbols for /lib/libc.so.7
Reading symbols from /lib/ld-elf.so.1...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-elf.so.1
#0  0x000000080095d0ea in kill () from /lib/libc.so.7
(gdb) bt
#0  0x000000080095d0ea in kill () from /lib/libc.so.7
#1  0x000000080095b819 in abort () from /lib/libc.so.7
#2  0x00000000004007aa in main ()
(gdb) f 2
#2  0x00000000004007aa in main ()
(gdb) i lo
No symbol table info available.
(gdb) i main
Undefined info command: "main".  Try "help info".
(gdb) disas main
Dump of assembler code for function main:
0x0000000000400780 <main+0>:    push   %rbp
0x0000000000400781 <main+1>:    mov    %rsp,%rbp
0x0000000000400784 <main+4>:    mov    $0x400816,%edi
0x0000000000400789 <main+9>:    mov    $0x1,%esi
0x000000000040078e <main+14>:   mov    $0x2,%edx
0x0000000000400793 <main+19>:   mov    $0x3,%ecx
0x0000000000400798 <main+24>:   mov    $0x4,%r8d
0x000000000040079e <main+30>:   xor    %eax,%eax
0x00000000004007a0 <main+32>:   callq  0x4004dc <printf@plt>
0x00000000004007a5 <main+37>:   callq  0x40050c <abort@plt>
0x00000000004007aa <main+42>:   nopw   0x0(%rax,%rax,1)
End of assembler dump.
(gdb) █
```

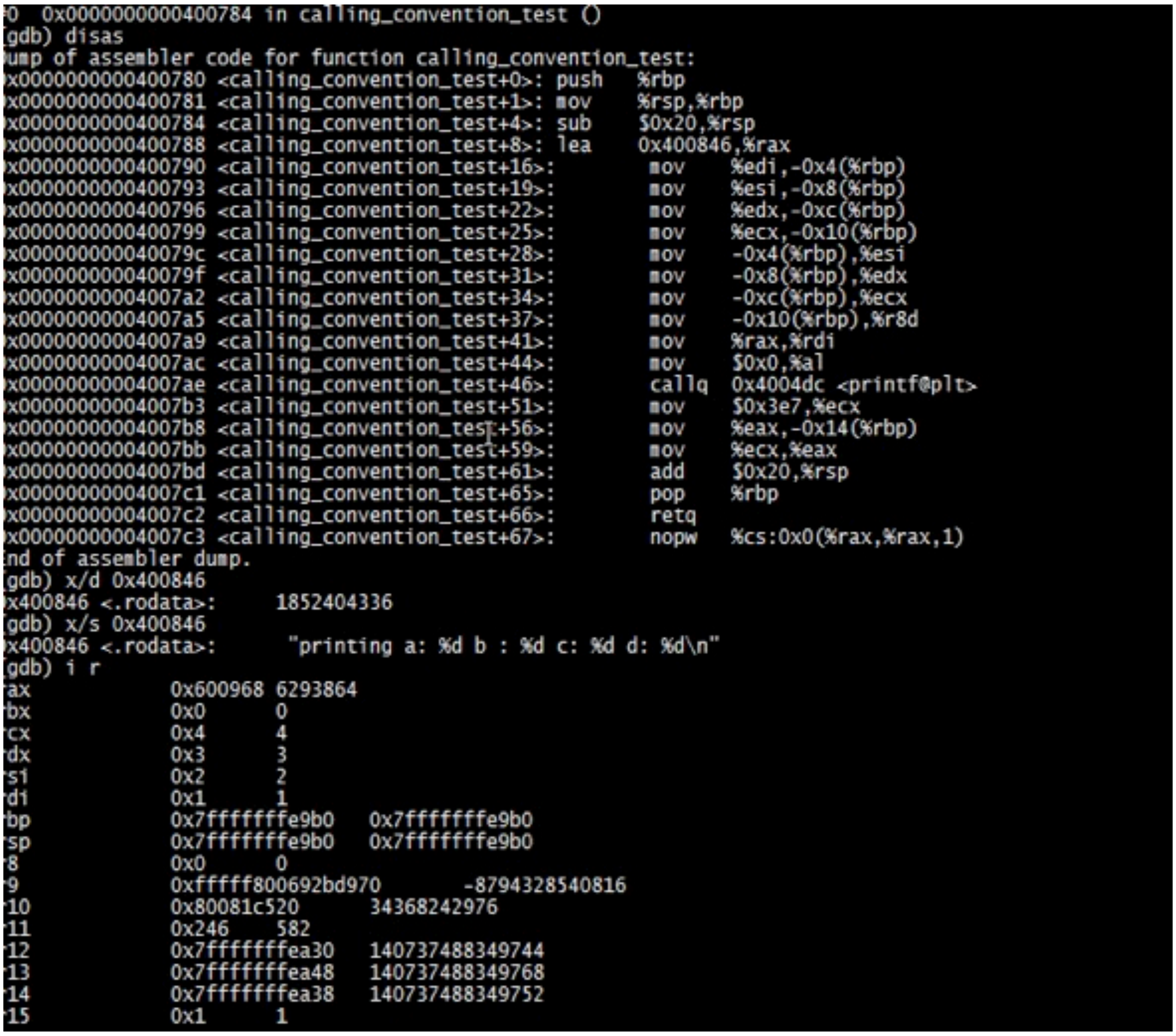
Figure 1. disas command on main



```
(gdb) disas calling_convention_test
Dump of assembler code for function calling_convention_test:
0x00000000004007b0 <calling_convention_test+0>: push   %rbp
0x00000000004007b1 <calling_convention_test+1>: mov    %rsp,%rbp
0x00000000004007b4 <calling_convention_test+4>: mov    %ecx,%r8d
0x00000000004007b7 <calling_convention_test+7>: mov    %edx,%ecx
0x00000000004007b9 <calling_convention_test+9>: mov    %esi,%edx
0x00000000004007bb <calling_convention_test+11>:        mov    %edi,%esi
0x00000000004007bd <calling_convention_test+13>:        mov    $0x400816,%edi
0x00000000004007c2 <calling_convention_test+18>:        xor    %eax,%eax
0x00000000004007c4 <calling_convention_test+20>:        callq  0x4004dc <printf@plt>
0x00000000004007c9 <calling_convention_test+25>:        mov    $0x3e7,%eax
0x00000000004007ce <calling_convention_test+30>:        pop    %rbp
0x00000000004007cf <calling_convention_test+31>:        retq
End of assembler dump.
(gdb) disas main 0x04007cf
Dump of assembler code from 0x400780 to 0x4007cf:
0x0000000000400780 <main+0>:    push   %rbp
0x0000000000400781 <main+1>:    mov    %rsp,%rbp
0x0000000000400784 <main+4>:    mov    $0x400816,%edi
0x0000000000400789 <main+9>:    mov    $0x1,%esi
0x000000000040078e <main+14>:   mov    $0x2,%edx
0x0000000000400793 <main+19>:   mov    $0x3,%ecx
0x0000000000400798 <main+24>:   mov    $0x4,%r8d
0x000000000040079e <main+30>:   xor    %eax,%eax
0x00000000004007a0 <main+32>:   callq  0x4004dc <printf@plt>
0x00000000004007a5 <main+37>:   callq  0x40050c <abort@plt>
0x00000000004007aa <main+42>:   nopw   0x0(%rax,%rax,1)
0x00000000004007b0 <calling_convention_test+0>: push   %rbp
0x00000000004007b1 <calling_convention_test+1>: mov    %rsp,%rbp
0x00000000004007b4 <calling_convention_test+4>: mov    %ecx,%r8d
0x00000000004007b7 <calling_convention_test+7>: mov    %edx,%ecx
0x00000000004007b9 <calling_convention_test+9>: mov    %esi,%edx
0x00000000004007bb <calling_convention_test+11>:        mov    %edi,%esi
0x00000000004007bd <calling_convention_test+13>:        mov    $0x400816,%edi
0x00000000004007c2 <calling_convention_test+18>:        xor    %eax,%eax
0x00000000004007c4 <calling_convention_test+20>:        callq  0x4004dc <printf@plt>
0x00000000004007c9 <calling_convention_test+25>:        mov    $0x3e7,%eax
0x00000000004007ce <calling_convention_test+30>:        pop    %rbp
End of assembler dump.
(gdb) █
```

Figure 2. disas command on calling_convention_test

Set a breakpoint at the calling_convention_test function, and run the program from the top.



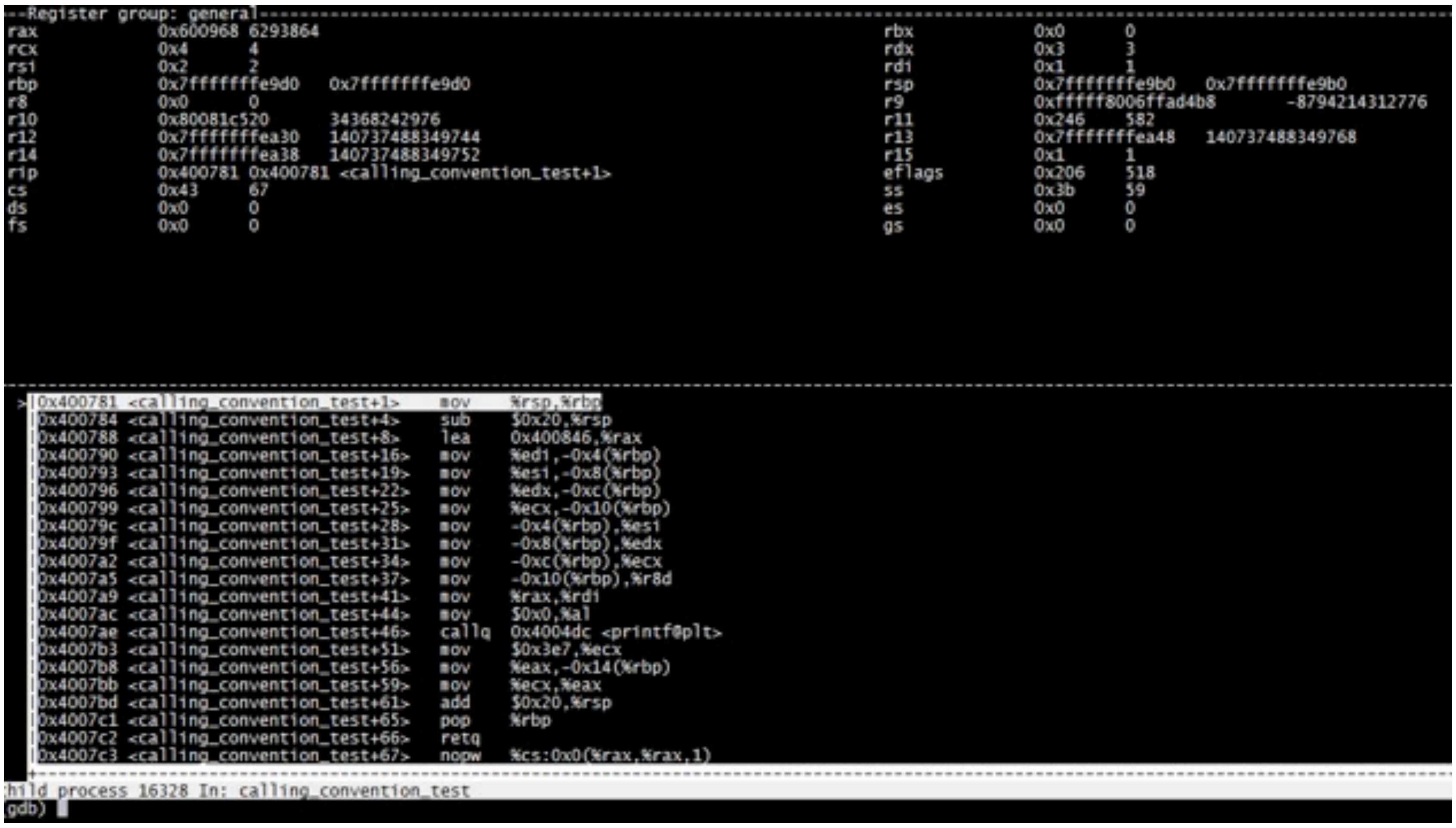
```
0  0x0000000000400784 in calling_convention_test ()
(gdb) disas
ump of assembler code for function calling_convention_test:
x0000000000400780 <calling_convention_test+0>: push   %rbp
x0000000000400781 <calling_convention_test+1>: mov    %rsp,%rbp
x0000000000400784 <calling_convention_test+4>: sub    $0x20,%rsp
x0000000000400788 <calling_convention_test+8>: lea    0x400846,%rax
x0000000000400790 <calling_convention_test+16>:        mov    %edi,-0x4(%rbp)
x0000000000400793 <calling_convention_test+19>:        mov    %esi,-0x8(%rbp)
x0000000000400796 <calling_convention_test+22>:        mov    %edx,-0xc(%rbp)
x0000000000400799 <calling_convention_test+25>:        mov    %ecx,-0x10(%rbp)
x000000000040079c <calling_convention_test+28>:        mov    -0x4(%rbp),%esi
x000000000040079f <calling_convention_test+31>:        mov    -0x8(%rbp),%edx
x00000000004007a2 <calling_convention_test+34>:        mov    -0xc(%rbp),%ecx
x00000000004007a5 <calling_convention_test+37>:        mov    -0x10(%rbp),%r8d
x00000000004007a9 <calling_convention_test+41>:        mov    %rax,%rdi
x00000000004007ac <calling_convention_test+44>:        mov    $0x0,%al
x00000000004007ae <calling_convention_test+46>:        callq  0x4004dc <printf@plt>
x00000000004007b3 <calling_convention_test+51>:        mov    $0x3e7,%ecx
x00000000004007b8 <calling_convention_test+56>:        mov    %eax,-0x14(%rbp)
x00000000004007bb <calling_convention_test+59>:        mov    %ecx,%eax
x00000000004007bd <calling_convention_test+61>:        add    $0x20,%rsp
x00000000004007c1 <calling_convention_test+65>:        pop    %rbp
x00000000004007c2 <calling_convention_test+66>:        retq
x00000000004007c3 <calling_convention_test+67>:        nopw   %cs:0x0(%rax,%rax,1)
nd of assembler dump.
(gdb) x/d 0x400846
x400846 <.rodata>:     1852404336
(gdb) x/s 0x400846
x400846 <.rodata>:      "printing a: %d b : %d c: %d d: %d\n"
(gdb) i r
ax             0x600968 6293864
bx             0x0      0
cx             0x4      4
dx             0x3      3
si             0x2      2
di             0x1      1
bp             0x7fffffffe9b0   0x7fffffffe9b0
sp             0x7fffffffe9b0   0x7fffffffe9b0
8              0x0      0
9              0xfffff800692bd970       -8794328540816
10             0x80081c520      34368242976
11             0x246    582
12             0x7fffffffea30   140737488349744
13             0x7fffffffea48   140737488349768
14             0x7fffffffea38   140737488349752
15             0x1      1
```

Figure 3. Using disas command

As you can see, our parameters in the registers are being taken from the stack. To see this in a friendlier interface, use the layout command:

```
(gdb) layout asm (gdb) layout regs
```

That will split the screen showing the registers as they change at the top of the screen, and the asm instructions at the bottom.



```
--Register group: general-----------------------------------------------------------------------------------------
rax            0x600968 6293864                                        rbx            0x0      0
rcx            0x4      4                                              rdx            0x3      3
rsi            0x2      2                                              rdi            0x1      1
rbp            0x7fffffffe9d0   0x7fffffffe9d0                         rsp            0x7fffffffe9b0   0x7fffffffe9b0
r8             0x0      0                                              r9             0xfffff8006ffad4b8       -8794214312776
r10            0x80081c520      34368242976                            r11            0x246    582
r12            0x7fffffffea30   140737488349744                        r13            0x7fffffffea48   140737488349768
r14            0x7fffffffea38   140737488349752                        r15            0x1      1
rip            0x400781 0x400781 <calling_convention_test+1>           eflags         0x206    518
cs             0x43     67                                             ss             0x3b     59
ds             0x0      0                                              es             0x0      0
fs             0x0      0                                              gs             0x0      0

-------------------------------------------------------------------------------------------------------------------
  >|0x400781 <calling_convention_test+1>    mov    %rsp,%rbp
   |0x400784 <calling_convention_test+4>    sub    $0x20,%rsp
   |0x400788 <calling_convention_test+8>    lea    0x400846,%rax
   |0x400790 <calling_convention_test+16>   mov    %edi,-0x4(%rbp)
   |0x400793 <calling_convention_test+19>   mov    %esi,-0x8(%rbp)
   |0x400796 <calling_convention_test+22>   mov    %edx,-0xc(%rbp)
   |0x400799 <calling_convention_test+25>   mov    %ecx,-0x10(%rbp)
   |0x40079c <calling_convention_test+28>   mov    -0x4(%rbp),%esi
   |0x40079f <calling_convention_test+31>   mov    -0x8(%rbp),%edx
   |0x4007a2 <calling_convention_test+34>   mov    -0xc(%rbp),%ecx
   |0x4007a5 <calling_convention_test+37>   mov    -0x10(%rbp),%r8d
   |0x4007a9 <calling_convention_test+41>   mov    %rax,%rdi
   |0x4007ac <calling_convention_test+44>   mov    $0x0,%al
   |0x4007ae <calling_convention_test+46>   callq  0x4004dc <printf@plt>
   |0x4007b3 <calling_convention_test+51>   mov    $0x3e7,%ecx
   |0x4007b8 <calling_convention_test+56>   mov    %eax,-0x14(%rbp)
   |0x4007bb <calling_convention_test+59>   mov    %ecx,%eax
   |0x4007bd <calling_convention_test+61>   add    $0x20,%rsp
   |0x4007c1 <calling_convention_test+65>   pop    %rbp
   |0x4007c2 <calling_convention_test+66>   retq
   |0x4007c3 <calling_convention_test+67>   nopw   %cs:0x0(%rax,%rax,1)
   +-----------------------------------------------------------------------------------------------------------------
hild process 16328 In: calling_convention_test
gdb)
```

Figure 4. Seeing machine code and register info at the same time

If you see the contents of the registers specified in the AMD64 ABI (%rdi,%rsi,%rdx,%rcx,%r8 and %r9), you will see the contents of the arguments that are being passed to the function <main+9>. If you only want to check the registers without using the GUI, just type:

```
(gdb) info regs
```

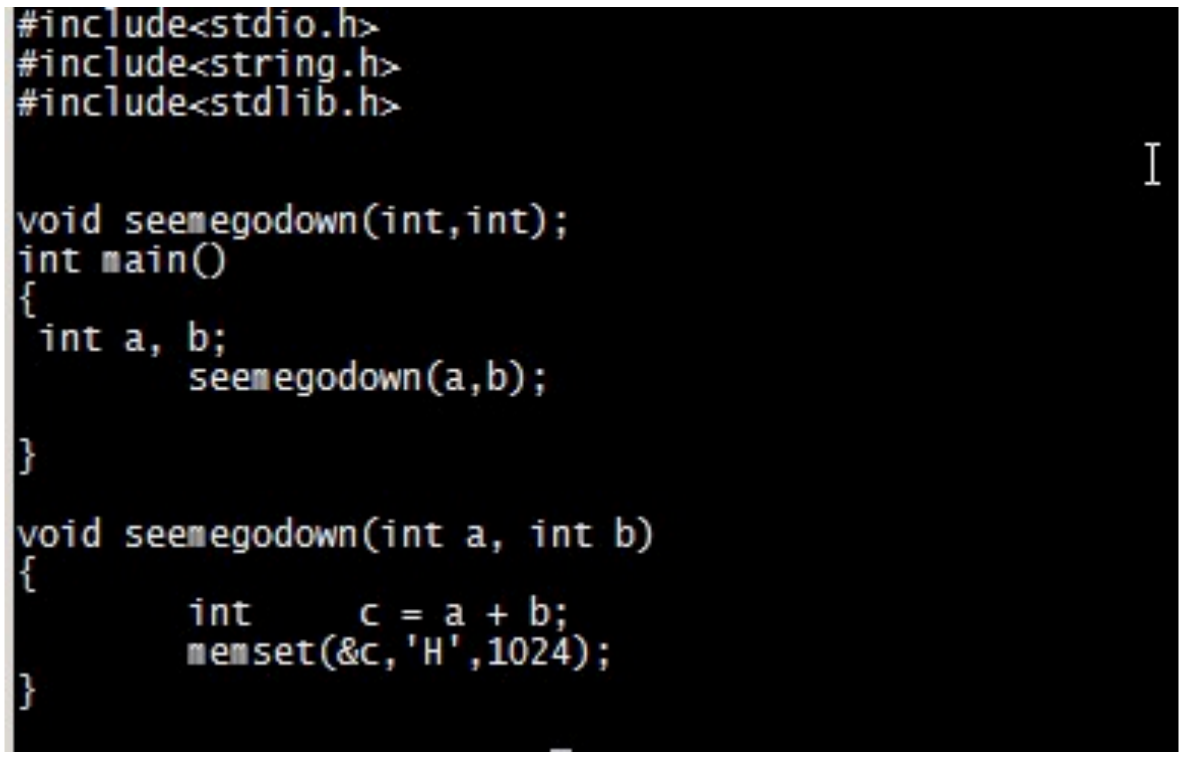
Or just print the one you need to check, for example %rcx register:

```
(gdb) p $rcx
```

What about if the executable has been stripped?

This is tougher to debug as we don't have the symbol tables available. The symbol tables have been stripped from the binary, which means, we do not have the function names, all references to the names are gone and we only have all the operations using memory addresses. That means, we cannot even set a breakpoint using main. However, all we need to know the address of our main function.

Let's take a look at another example program:



```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>


void seemegodown(int,int);
int main()
{
 int a, b;
        seemegodown(a,b);

}

void seemegodown(int a, int b)
{
        int     c = a + b;
        memset(&c,'H',1024);
}
```

Figure 5. Code example 2

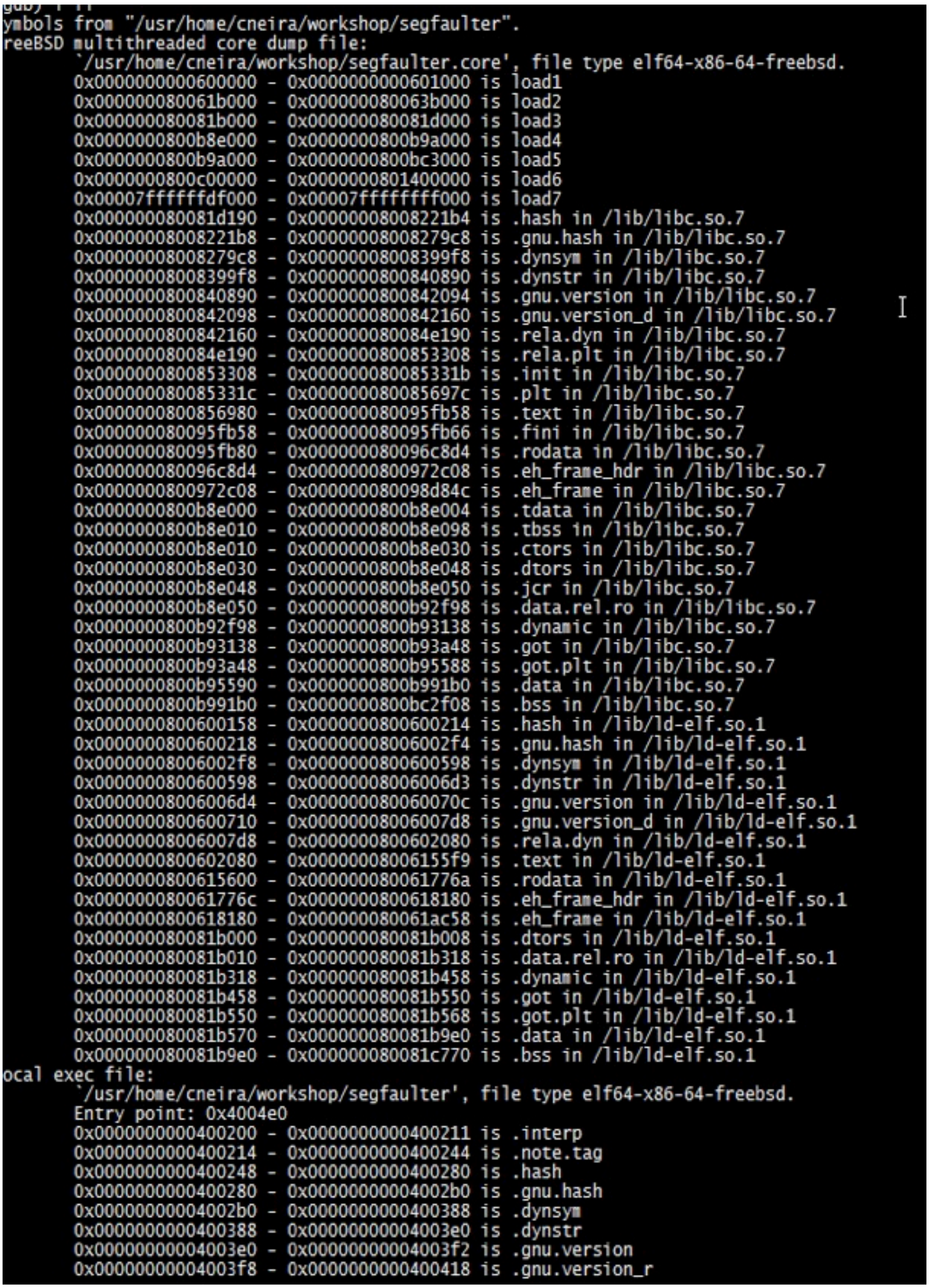
Compile it as usual, and strip it:

```
$ cc segfaulter.c -o segfaulter
```

```
$ strip segfaulter
```

Now, let's find where the main is at.

```
(gdb) info file
```



```
gdb) i f
ymbols from "/usr/home/cneira/workshop/segfaulter".
reeBSD multithreaded core dump file:
        `/usr/home/cneira/workshop/segfaulter.core', file type elf64-x86-64-freebsd.
        0x0000000000600000 - 0x0000000000601000 is load1
        0x000000080061b000 - 0x000000080063b000 is load2
        0x000000080081b000 - 0x000000080081d000 is load3
        0x0000000800b8e000 - 0x0000000800b9a000 is load4
        0x0000000800b9a000 - 0x0000000800bc3000 is load5
        0x0000000800c00000 - 0x0000000801400000 is load6
        0x00007ffffffdf000 - 0x00007ffffffff000 is load7
        0x000000080081d190 - 0x00000008008221b4 is .hash in /lib/libc.so.7
        0x00000008008221b8 - 0x00000008008279c8 is .gnu.hash in /lib/libc.so.7
        0x00000008008279c8 - 0x00000008008399f8 is .dynsym in /lib/libc.so.7
        0x00000008008399f8 - 0x0000000800840890 is .dynstr in /lib/libc.so.7
        0x0000000800840890 - 0x0000000800842094 is .gnu.version in /lib/libc.so.7
        0x0000000800842098 - 0x0000000800842160 is .gnu.version_d in /lib/libc.so.7
        0x0000000800842160 - 0x000000080084e190 is .rela.dyn in /lib/libc.so.7
        0x000000080084e190 - 0x0000000800853308 is .rela.plt in /lib/libc.so.7
        0x0000000800853308 - 0x000000080085331b is .init in /lib/libc.so.7
        0x000000080085331c - 0x000000080085697c is .plt in /lib/libc.so.7
        0x0000000800856980 - 0x000000080095fb58 is .text in /lib/libc.so.7
        0x000000080095fb58 - 0x000000080095fb66 is .fini in /lib/libc.so.7
        0x000000080095fb80 - 0x000000080096c8d4 is .rodata in /lib/libc.so.7
        0x000000080096c8d4 - 0x0000000800972c08 is .eh_frame_hdr in /lib/libc.so.7
        0x0000000800972c08 - 0x000000080098d84c is .eh_frame in /lib/libc.so.7
        0x0000000800b8e000 - 0x0000000800b8e004 is .tdata in /lib/libc.so.7
        0x0000000800b8e010 - 0x0000000800b8e098 is .tbss in /lib/libc.so.7
        0x0000000800b8e010 - 0x0000000800b8e030 is .ctors in /lib/libc.so.7
        0x0000000800b8e030 - 0x0000000800b8e048 is .dtors in /lib/libc.so.7
        0x0000000800b8e048 - 0x0000000800b8e050 is .jcr in /lib/libc.so.7
        0x0000000800b8e050 - 0x0000000800b92f98 is .data.rel.ro in /lib/libc.so.7
        0x0000000800b92f98 - 0x0000000800b93138 is .dynamic in /lib/libc.so.7
        0x0000000800b93138 - 0x0000000800b93a48 is .got in /lib/libc.so.7
        0x0000000800b93a48 - 0x0000000800b95588 is .got.plt in /lib/libc.so.7
        0x0000000800b95590 - 0x0000000800b991b0 is .data in /lib/libc.so.7
        0x0000000800b991b0 - 0x0000000800bc2f08 is .bss in /lib/libc.so.7
        0x0000000800600158 - 0x0000000800600214 is .hash in /lib/ld-elf.so.1
        0x0000000800600218 - 0x00000008006002f4 is .gnu.hash in /lib/ld-elf.so.1
        0x00000008006002f8 - 0x0000000800600598 is .dynsym in /lib/ld-elf.so.1
        0x0000000800600598 - 0x00000008006006d3 is .dynstr in /lib/ld-elf.so.1
        0x00000008006006d4 - 0x000000080060070c is .gnu.version in /lib/ld-elf.so.1
        0x0000000800600710 - 0x00000008006007d8 is .gnu.version_d in /lib/ld-elf.so.1
        0x00000008006007d8 - 0x0000000800602080 is .rela.dyn in /lib/ld-elf.so.1
        0x0000000800602080 - 0x00000008006155f9 is .text in /lib/ld-elf.so.1
        0x0000000800615600 - 0x000000080061776a is .rodata in /lib/ld-elf.so.1
        0x000000080061776c - 0x0000000800618180 is .eh_frame_hdr in /lib/ld-elf.so.1
        0x0000000800618180 - 0x000000080061ac58 is .eh_frame in /lib/ld-elf.so.1
        0x000000080081b000 - 0x000000080081b008 is .dtors in /lib/ld-elf.so.1
        0x000000080081b010 - 0x000000080081b318 is .data.rel.ro in /lib/ld-elf.so.1
        0x000000080081b318 - 0x000000080081b458 is .dynamic in /lib/ld-elf.so.1
        0x000000080081b458 - 0x000000080081b550 is .got in /lib/ld-elf.so.1
        0x000000080081b550 - 0x000000080081b568 is .got.plt in /lib/ld-elf.so.1
        0x000000080081b570 - 0x000000080081b9e0 is .data in /lib/ld-elf.so.1
        0x000000080081b9e0 - 0x000000080081c770 is .bss in /lib/ld-elf.so.1
ocal exec file:
        `/usr/home/cneira/workshop/segfaulter', file type elf64-x86-64-freebsd.
        Entry point: 0x4004e0
        0x0000000000400200 - 0x0000000000400211 is .interp
        0x0000000000400214 - 0x0000000000400244 is .note.tag
        0x0000000000400248 - 0x0000000000400280 is .hash
        0x0000000000400280 - 0x00000000004002b0 is .gnu.hash
        0x00000000004002b0 - 0x0000000000400388 is .dynsym
        0x0000000000400388 - 0x00000000004003e0 is .dynstr
        0x00000000004003e0 - 0x00000000004003f2 is .gnu.version
        0x00000000004003f8 - 0x0000000000400418 is .gnu.version_r
```

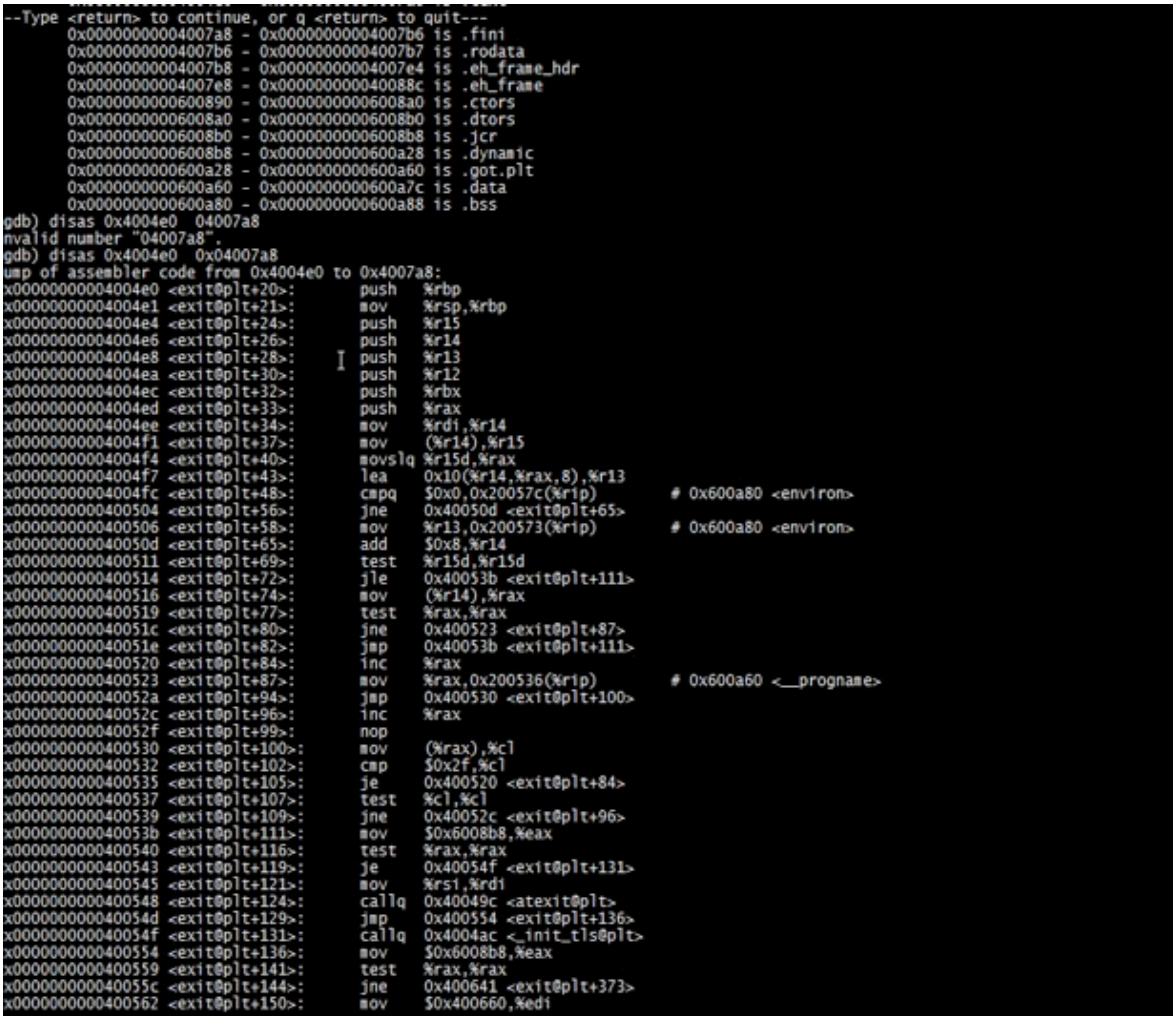
Figure 6. Info file command

There is the Entry point: 0x4004e0

Let’s set a breakpoint at that memory area: (gdb) b *0x4004e0

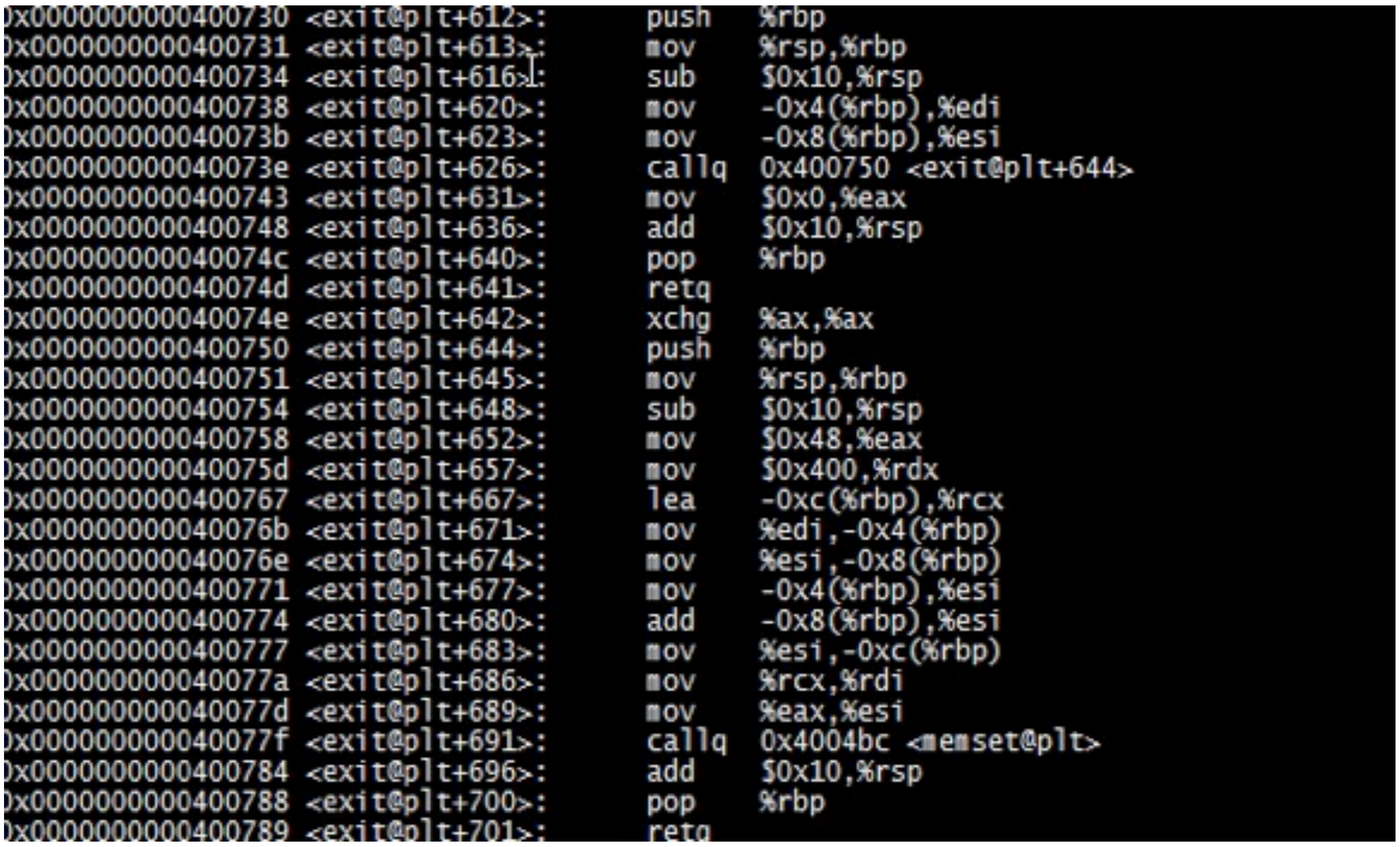
Then, as we know that the program text area (code) is between 0x4004e0 and 0x4007a8. We could disassemble the range between those addresses (not really a thing you will want to do in a real application) other option that is nicer is just to display the program counter $pc the next couple of instructions every time you go down one instruction (nexti).

```
(gdb) display /i $pc
```



```
--Type <return> to continue, or q <return> to quit---
        0x00000000004007a8 - 0x00000000004007b6 is .fini
        0x00000000004007b6 - 0x00000000004007b7 is .rodata
        0x00000000004007b8 - 0x00000000004007e4 is .eh_frame_hdr
        0x00000000004007e8 - 0x000000000040088c is .eh_frame
        0x0000000000600890 - 0x00000000006008a0 is .ctors
        0x00000000006008a0 - 0x00000000006008b0 is .dtors
        0x00000000006008b0 - 0x00000000006008b8 is .jcr
        0x00000000006008b8 - 0x0000000000600a28 is .dynamic
        0x0000000000600a28 - 0x0000000000600a60 is .got.plt
        0x0000000000600a60 - 0x0000000000600a7c is .data
        0x0000000000600a80 - 0x0000000000600a88 is .bss
gdb) disas 0x4004e0  04007a8
nvalid number "04007a8".
gdb) disas 0x4004e0  0x04007a8
ump of assembler code from 0x4004e0 to 0x4007a8:
x00000000004004e0 <exit@plt+20>:        push   %rbp
x00000000004004e1 <exit@plt+21>:        mov    %rsp,%rbp
x00000000004004e4 <exit@plt+24>:        push   %r15
x00000000004004e6 <exit@plt+26>:        push   %r14
x00000000004004e8 <exit@plt+28>:        push   %r13
x00000000004004ea <exit@plt+30>:        push   %r12
x00000000004004ec <exit@plt+32>:        push   %rbx
x00000000004004ed <exit@plt+33>:        push   %rax
x00000000004004ee <exit@plt+34>:        mov    %rdi,%r14
x00000000004004f1 <exit@plt+37>:        mov    (%r14),%r15
x00000000004004f4 <exit@plt+40>:        movslq %r15d,%rax
x00000000004004f7 <exit@plt+43>:        lea    0x10(%r14,%rax,8),%r13
x00000000004004fc <exit@plt+48>:        cmpq   $0x0,0x20057c(%rip)        # 0x600a80 <environ>
x0000000000400504 <exit@plt+56>:        jne    0x40050d <exit@plt+65>
x0000000000400506 <exit@plt+58>:        mov    %r13,0x200573(%rip)        # 0x600a80 <environ>
x000000000040050d <exit@plt+65>:        add    $0x8,%r14
x0000000000400511 <exit@plt+69>:        test   %r15d,%r15d
x0000000000400514 <exit@plt+72>:        jle    0x40053b <exit@plt+111>
x0000000000400516 <exit@plt+74>:        mov    (%r14),%rax
x0000000000400519 <exit@plt+77>:        test   %rax,%rax
x000000000040051c <exit@plt+80>:        jne    0x400523 <exit@plt+87>
x000000000040051e <exit@plt+82>:        jmp    0x40053b <exit@plt+111>
x0000000000400520 <exit@plt+84>:        inc    %rax
x0000000000400523 <exit@plt+87>:        mov    %rax,0x200536(%rip)        # 0x600a60 <__progname>
x000000000040052a <exit@plt+94>:        jmp    0x400530 <exit@plt+100>
x000000000040052c <exit@plt+96>:        inc    %rax
x000000000040052f <exit@plt+99>:        nop
x0000000000400530 <exit@plt+100>:       mov    (%rax),%cl
x0000000000400532 <exit@plt+102>:       cmp    $0x2f,%cl
x0000000000400535 <exit@plt+105>:       je     0x400520 <exit@plt+84>
x0000000000400537 <exit@plt+107>:       test   %cl,%cl
x0000000000400539 <exit@plt+109>:       jne    0x40052c <exit@plt+96>
x000000000040053b <exit@plt+111>:       mov    $0x6008b8,%eax
x0000000000400540 <exit@plt+116>:       test   %rax,%rax
x0000000000400543 <exit@plt+119>:       je     0x40054f <exit@plt+131>
x0000000000400545 <exit@plt+121>:       mov    %rsi,%rdi
x0000000000400548 <exit@plt+124>:       callq  0x40049c <atexit@plt>
x000000000040054d <exit@plt+129>:       jmp    0x400554 <exit@plt+136>
x000000000040054f <exit@plt+131>:       callq  0x4004ac <_init_tls@plt>
x0000000000400554 <exit@plt+136>:       mov    $0x6008b8,%eax
x0000000000400559 <exit@plt+141>:       test   %rax,%rax
x000000000040055c <exit@plt+144>:       jne    0x400641 <exit@plt+373>
x0000000000400562 <exit@plt+150>:       mov    $0x400660,%edi
```

Figure 7. disas command addresses range



```
x0000000000400730 <exit@plt+612>:     push   %rbp
x0000000000400731 <exit@plt+613>:     mov    %rsp,%rbp
x0000000000400734 <exit@plt+616>:     sub    $0x10,%rsp
x0000000000400738 <exit@plt+620>:     mov    -0x4(%rbp),%edi
x000000000040073b <exit@plt+623>:     mov    -0x8(%rbp),%esi
x000000000040073e <exit@plt+626>:     callq  0x400750 <exit@plt+644>
x0000000000400743 <exit@plt+631>:     mov    $0x0,%eax
x0000000000400748 <exit@plt+636>:     add    $0x10,%rsp
x000000000040074c <exit@plt+640>:     pop    %rbp
x000000000040074d <exit@plt+641>:     retq
x000000000040074e <exit@plt+642>:     xchg   %ax,%ax
x0000000000400750 <exit@plt+644>:     push   %rbp
x0000000000400751 <exit@plt+645>:     mov    %rsp,%rbp
x0000000000400754 <exit@plt+648>:     sub    $0x10,%rsp
x0000000000400758 <exit@plt+652>:     mov    $0x48,%eax
x000000000040075d <exit@plt+657>:     mov    $0x400,%rdx
x0000000000400767 <exit@plt+667>:     lea    -0xc(%rbp),%rcx
x000000000040076b <exit@plt+671>:     mov    %edi,-0x4(%rbp)
x000000000040076e <exit@plt+674>:     mov    %esi,-0x8(%rbp)
x0000000000400771 <exit@plt+677>:     mov    -0x4(%rbp),%esi
x0000000000400774 <exit@plt+680>:     add    -0x8(%rbp),%esi
x0000000000400777 <exit@plt+683>:     mov    %esi,-0xc(%rbp)
x000000000040077a <exit@plt+686>:     mov    %rcx,%rdi
x000000000040077d <exit@plt+689>:     mov    %eax,%esi
x000000000040077f <exit@plt+691>:     callq  0x4004bc <memset@plt>
x0000000000400784 <exit@plt+696>:     add    $0x10,%rsp
x0000000000400788 <exit@plt+700>:     pop    %rbp
x0000000000400789 <exit@plt+701>:     retq
```

Here is our function (notice the call to memset in there) <exit@plt+644>.

Useful commands for looking at machine code:

Apart for the ones described earlier, you can use these ones to take a look at what is happening in your program.

Examining memory

Use the **x** command to examine memory. The syntax for the x command is x/FMT ADDRESS. The FMT field is a count followed by a format letter and a size letter. Use the help command 'help x' to see them all. The ADDRESS argument can either be a symbol name, such as a variable, or a memory address.

Examine the variable as a string:

```
(gdb) x/s <var>
```

Examine the variable as a character:

```
(gdb) x/c <var>
```

Examine the variable as 4 characters:

```
(gdb) x/4c <var>
```

Examine the first 32 bits of the variable:

```
(gdb) x/t <var>
```

Examine the first 24 bytes of the variable in hex:

```
(gdb) x/3x <var>
```

Print the instructions at the memory area `specified (gdb) x/i  <addr>`

Stepping

As we inspect the machine code, we don’t have the source code. The commands that we usually use: next (n) and step(s) have their equivalents named with the ‘I’ suffix (for instruction):

nexti (ni)

stepi (si)

Question:

Based on the documentation, describe the GDB session steps that you will use to check the parameters being passed to the memset function, and the user-defined function of the program on example program 2.

Introduction to Dtrace

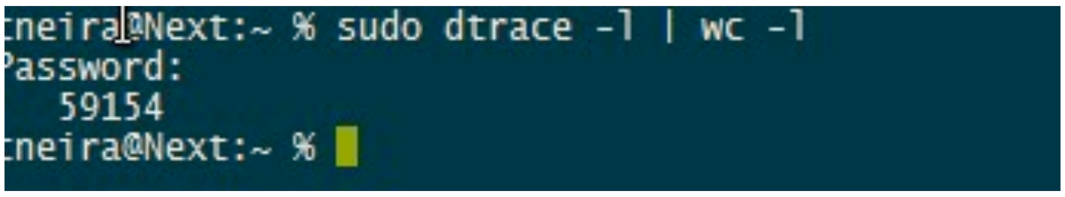
DTrace, or dynamic tracing, was first available in the Solaris 10 3/05 around 2005. DTrace is now available in FreeBSD beginning from 7.1 and Mac OS X from 10.5 (Leopard). DTrace differs from traditional tools in that code is instrumented dynamically (that means you can peek at the program without recompiling).

From the handbook: https://www.freebsd.org/doc/en_US.ISO8859- 1/books/handbook/dtrace.html

"The FreeBSD implementation provides full support for kernel DTrace and experimental support for userland DTrace. Userland DTrace allows users to perform function boundary tracing for userland programs using the pid provider, and to insert static probes into userland programs for later tracing. Some ports, such as databases/postgres-server and lang/php5 have a DTrace option to enable static probes. FreeBSD 10.0-RELEASE has reasonably good userland DTrace support, but it is not considered production ready. In particular, it is possible to crash traced programs."

Requirements

I´m running FreeBSD 10.0-STABLE where *Dtrace* is already available as a kernel module. Typing the following as root will let you know that you are ready to fire up some probes using dtrace:



```
cneira@Next:~ % sudo dtrace -l | wc -l
Password:
   59154
cneira@Next:~ %
```

If this fails, you need to recompile your kernel and follow the instructions from the handbook in here: https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/dtrace-enable.html

Why do I care about DTrace?

If you want to understand what is happening in your software without needing recompiled special versions of your applications (lots of debug messages, maybe recompile with debug flags to use a debugger?) and also centralize all your current instrumentation tools into just one, then you should care about *DTrace.*

Some Features of DTrace.

DTrace is dynamic: probes are enabled only when you need them.

No code is present for inactive probes.

There is no performance degradation when you are not using DTrace.

When the dtrace command exits, all probes are disabled and instrumentation removed.

The system is reverted to its original state.

DTrace is nondestructive. The system is not paused or quiesced.

DTrace is designed to be efficient. No extra data are ever traced.

Because of its safety and efficiency, DTrace can be used in production to solve real problems in real time.

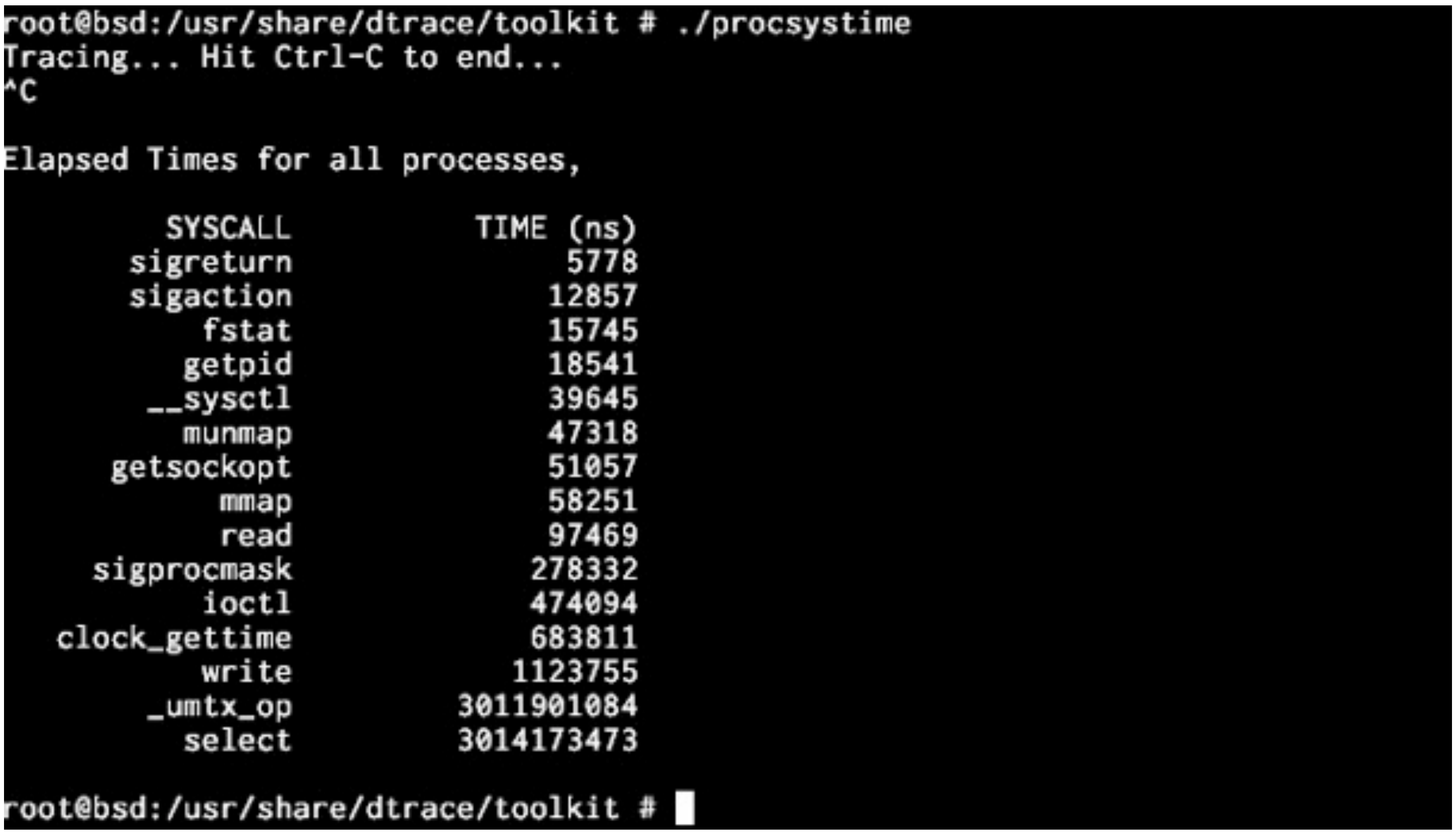
Predicates: A logical predicate mechanism allows actions to be taken only when user- specified conditions are met. Unwanted data is discarded at the source—never retained, copied, or stored.

A high-level control language: DTrace is equipped with an expressive C-like scripting language known as D . It supports all ANSI C operators, which may be familiar to you and reduce your learning curve, and allows access to the kernel's variables and native types. D offers user-defined variables, including global variables, thread-local variables, and associative arrays, and it supports pointer dereferencing. This, coupled with the runtime safety mechanisms of DTrace, makes it admirable

Trying DTrace :

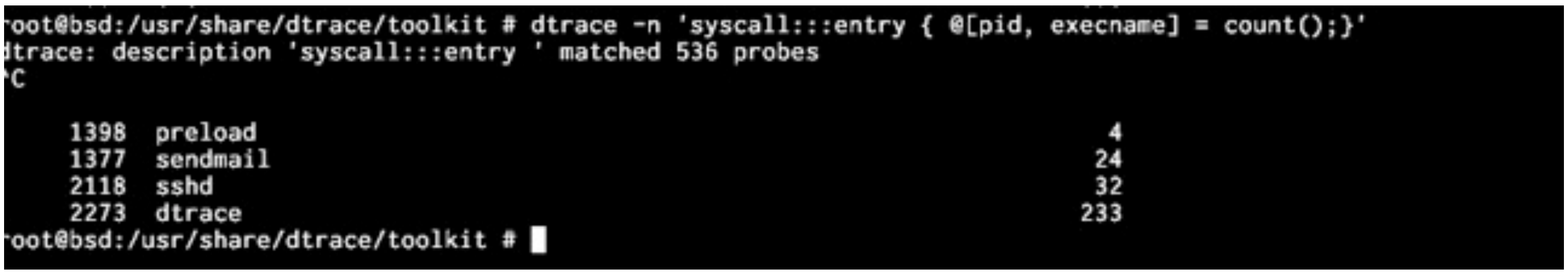
We will try a default script that comes with our FreeBSD installation. Go to

/usr/share/dtrace/toolkit and execute the script called procsystime. This script "only processes system call time details." Notice the only.



```
root@bsd:/usr/share/dtrace/toolkit # ./procsystime
Tracing... Hit Ctrl-C to end...
^C

Elapsed Times for all processes,

         SYSCALL          TIME (ns)
       sigreturn               5778
       sigaction              12857
           fstat              15745
          getpid              18541
        __sysctl              39645
          munmap              47318
      getsockopt              51057
            mmap              58251
            read              97469
     sigprocmask             278332
           ioctl             474094
   clock_gettime             683811
           write            1123755
        _umtx_op         3011901084
          select         3014173473

root@bsd:/usr/share/dtrace/toolkit # 
```

This one is pretty handy right away. Imagine what you could do with some imagination. Let´s see this one liner: which processes are executing the most system calls?



```
oot@bsd:/usr/share/dtrace/toolkit # dtrace -n 'syscall:::entry { @[pid, execname] = count();}'
trace: description 'syscall:::entry ' matched 536 probes
C

    1398  preload                                                          4
    1377  sendmail                                                        24
    2118  sshd                                                            32
    2273  dtrace                                                         233
oot@bsd:/usr/share/dtrace/toolkit # █
```

Pretty impressive, isn’t it? At this moment, you are wondering how all these years you have lived without DTrace. Using truss, strace ,lsof even gdb now seems pretty lame. Well, gdb is not so lame now.

DTrace Scripting:

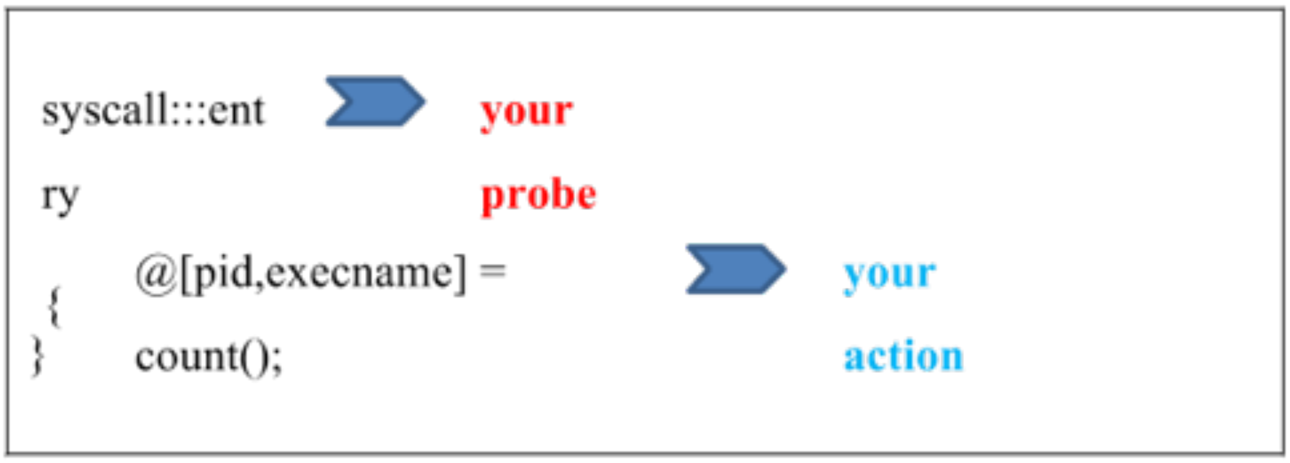
DTrace scripts are written in the D language. You could take a look at this reference http://dlang.org/spec.html.

Now, let´s write our first probe. A DTrace script has the following structure:

```
 Your probes

/ predicate (usually you will create a filter here) /

{

What are you going to do when you hit a probe

}
```

Let´s create a simple one to get used to the syntax, and later, we will dissect it line by line, ok? This one does not have a predicate, so it will capture all that the probe is asking for.

A predicate is a conditional statement (IF statement if you like).



Save this to a file called example1.d . Then, execute the script by typing: dtrace –s example1.d

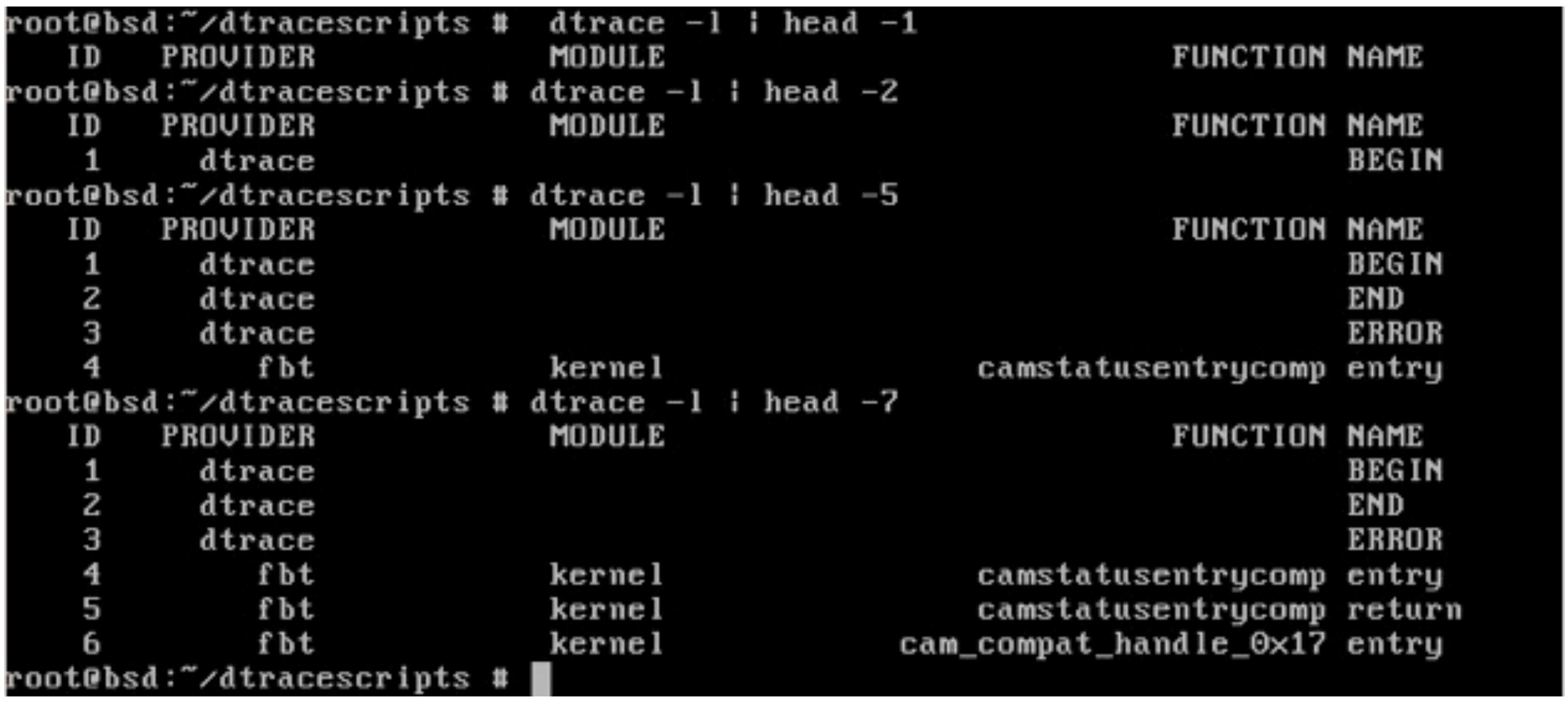
The probe section has the following syntax:

`provider:module:function:name`

What every section means:

Provider	The name of the DTrace provider that is publishing this probe. The provider name typically corresponds to the name of the DTrace kernel module that performs the instrumentation to enable the probe.
Module	If this probe corresponds to a specific program location, the name of the module in which the probe is located. This name is either the name of a kernel module or the name of a user library.
Function	If this probe corresponds to a specific program location, the name of the program function in which the probe is located.
Name	The final component of the probe name is a name that gives you some idea of the probe's semantic meaning, such as BEGIN or ENDS. In this case, the probe says that is the entry of a function call.

What providers are available to us in FreeBSD? Well, you should dig in and see what you need.



```
root@bsd:~/dtracescripts #  dtrace -l | head -1
   ID   PROVIDER            MODULE                          FUNCTION NAME
root@bsd:~/dtracescripts # dtrace -l | head -2
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
root@bsd:~/dtracescripts # dtrace -l | head -5
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4        fbt            kernel                camstatusentrycomp entry
root@bsd:~/dtracescripts # dtrace -l | head -7
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4        fbt            kernel                camstatusentrycomp entry
    5        fbt            kernel                camstatusentrycomp return
    6        fbt            kernel            cam_compat_handle_0x17 entry
root@bsd:~/dtracescripts #
```

Now to this line:

```
@[pid,execname] = count();
```

This is called an aggregation, and is denoted by the @ special character. Aggregations are global in your Dtrace scripts. The syntax of an aggregation is as indicated below:

```
@name[ keys ] = aggfunc ( args );
```

Name: The name you choose for the aggregation.

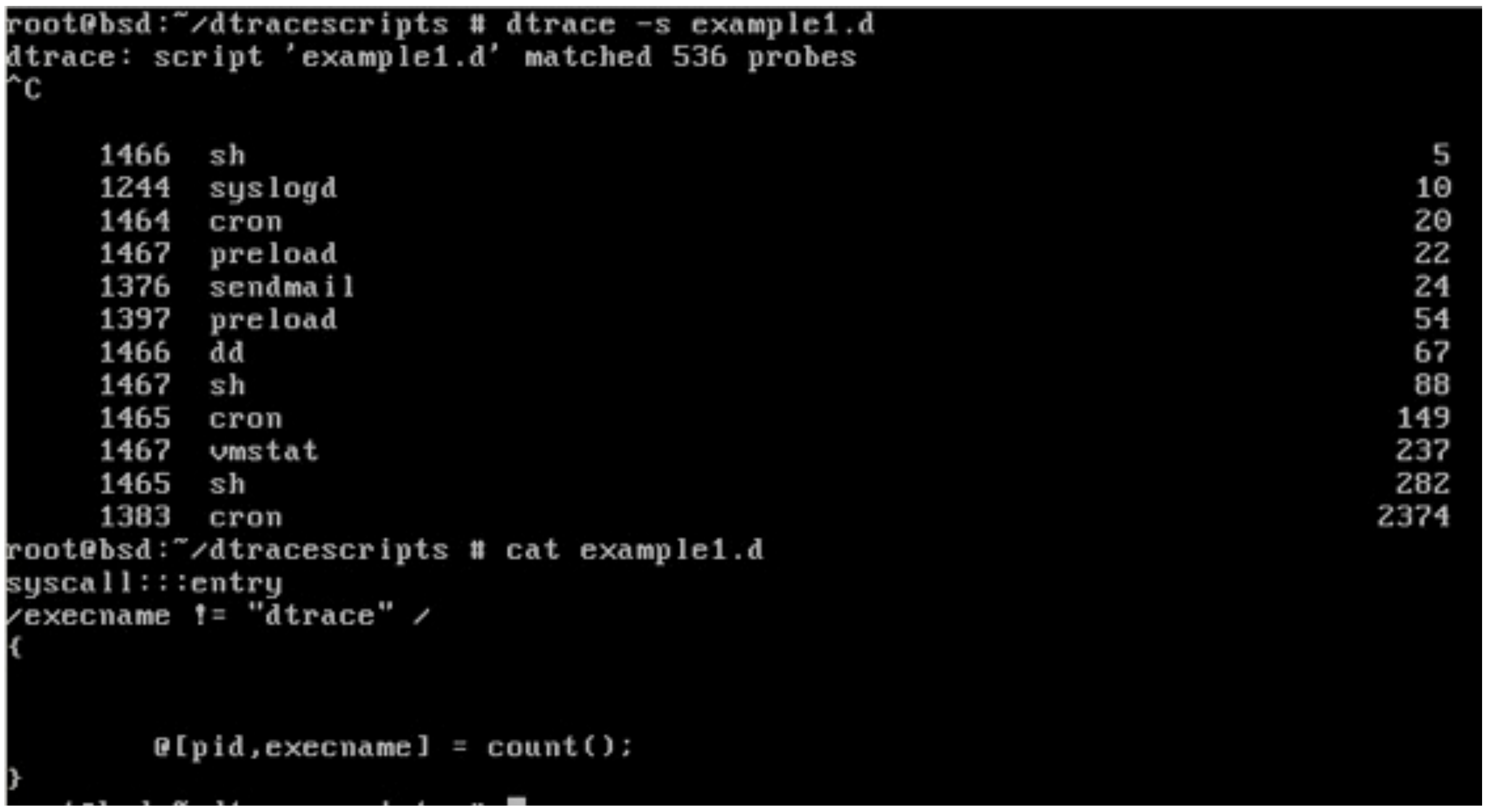
Keys: Comma-separated list of D expressions (in this case, we are asking for *pid* and the name of executable which triggers the probe).

Aggfunc: This is one of the DTrace aggregating functions. Moreover, args is a comma-separated list of arguments appropriate for the aggregating function.

Here are the aggregation functions available:

Function Name	Arguments	Result
count	none	The number of times called.
sum	scalar expression	The total value of the specified expressions.
avg	scalar expression	The arithmetic average of the specified expressions.
min	scalar expression	The smallest value among the specified expressions.
max	scalar expression	The largest value among the specified expressions.
lquantize	scalar expression, lower bound, upper bound, step value	A linear frequency distribution, sized by the specified range, of the values of the specified expressions. Increments the value in the **highest** bucket that is **less** than the specified expression.
quantize	scalar expression	A power-of-two frequency distribution of the values of the specified expressions. Increments the value in the **highest** power-of-two bucket that is **less** than the specified expression.

Now, let's add a predicate to the same script. If you looked at the output of the script you would have seen that it counted the system calls done by dtrace itself. Let's filter that.



```
root@bsd:~/dtracescripts # dtrace -s example1.d
dtrace: script 'example1.d' matched 536 probes
^C

     1466  sh                                                               5
     1244  syslogd                                                         10
     1464  cron                                                            20
     1467  preload                                                         22
     1376  sendmail                                                        24
     1397  preload                                                         54
     1466  dd                                                              67
     1467  sh                                                              88
     1465  cron                                                           149
     1467  vmstat                                                         237
     1465  sh                                                             282
     1383  cron                                                          2374
root@bsd:~/dtracescripts # cat example1.d
syscall:::entry
/execname != "dtrace" /
{


        @[pid,execname] = count();
}
```

But how did I know that *execname* contained the name of the program being executed? Well, it is a built-in variable in DTrace. Here is a list of some Dtrace Built-in variables:

DTrace Built-in Variables

Type and Name	Description
int64_t arg0, ..., arg9	The first ten input arguments to a probe represented as raw 64-bit integers. If fewer than ten arguments are passed to the current probe, the remaining variables return zero.
args[]	The typed arguments to the current probe, if any. The args[] array is accessed using an integer index, but each element is defined to be the type corresponding to the given probe argument. For example, if args[] is referenced by a read(2) system call probe, args[0] is of type int, args[1] is of type void *, and args[2] is of type size_t.
uintptr_t caller	The program counter location of the current thread just before entering the current probe.
string cwd	The name of the current working directory of the process associated with the current thread.
uint_t epid	The enabled probe ID (EPID) for the current probe. This integer uniquely identifiers a particular probe that is enabled with a specific predicate and set of actions.
int errno	The error value returned by the last system call executed by this thread.

	computations.
uid_t uid	The real user ID of the current process.
uint64_t uregs[]	The current thread's saved user-mode register values at probe firing time. Use of the uregs[].
uint64_t vmregs[]	The current thread's active virtual machine register values at probe firing time. Use of the vmregs[].
uint64_t vtimestamp	The current value of a nanosecond timestamp counter that is virtualized to the amount of time that the current thread has been running on a CPU, minus the time spent in DTrace predicates and actions. This counter increments from an arbitrary point in the past and should only be used for relative time computations.
uint64_t walltimestamp	The current number of nanoseconds since 00:00 Universal Coordinated Time, January 1, 1970.

You could take a look at the full listing in this URL:
http://docs.oracle.com/cd/E18752_01/html/819-5488/gcfpz.html

Assignment:

Pick any program that you are certain that uses the *strcmp*(3) C library function, and print the arguments passed to this function. Tip: first create a script, similar to the example, to check for the *strcmp* call being used in any the programs executing in your machine.

Carlos Antonio Neira Bustos

email: cneirabustos@gmail.com